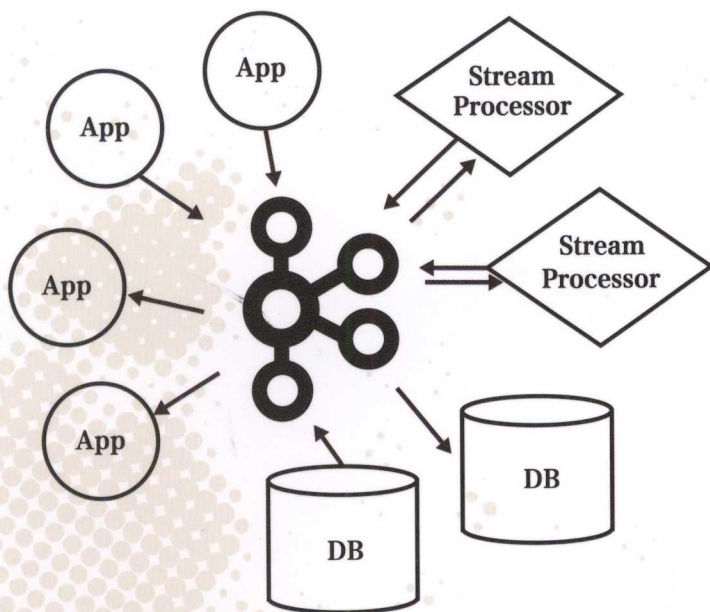



版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Apache Kafka 源码剖析

徐郡明 编著



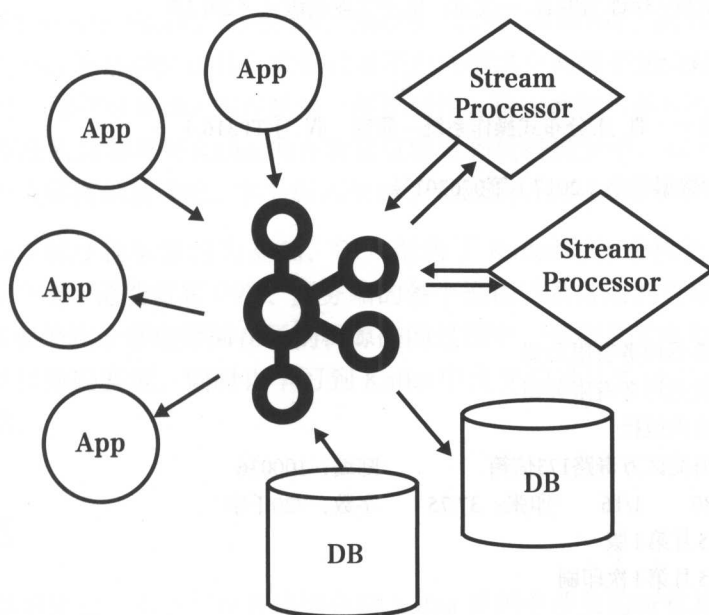


作者简介

徐郡明，武汉大学硕士，目前就职于航天科技集团旗下某研究所，主要负责政企云平台基础架构的设计和研发工作，有多年Kafka应用和设计经验。长期关注大数据处理相关技术以及Kafka的发展。

Apache Kafka 源码剖析

徐郡明 编著



电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书以 Kafka 0.10.0 版本源码为基础, 针对 Kafka 的架构设计到实现细节进行详细阐述。本书共 5 章, 从 Kafka 的应用场景、源码环境搭建开始逐步深入, 不仅介绍 Kafka 的核心概念, 而且对 Kafka 生产者、消费者、服务端的源码进行深入的剖析, 最后介绍 Kafka 常用的管理脚本实现, 让读者不仅从宏观设计上了解 Kafka, 而且能够深入到 Kafka 的细节设计之中。在源码分析的过程中, 还穿插了笔者工作积累的经验和对 Kafka 设计的理解, 希望读者可以举一反三, 不仅知其然, 而且知其所以然。

本书旨在为读者阅读 Kafka 源码提供帮助和指导, 让读者更加深入地了解 Kafka 的运行原理、设计理念, 让读者在设计分布式系统时可以参考 Kafka 的优秀设计。本书的内容对于读者全面提升自己的技术能力有很大帮助。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有, 侵权必究。

图书在版编目 (CIP) 数据

Apache Kafka 源码剖析 / 徐郡明编著. —北京: 电子工业出版社, 2017.5

ISBN 978-7-121-31345-5

I. ①A… II. ①徐… III. ①分布式操作系统—研究 IV. ①TP316.4

中国版本图书馆 CIP 数据核字 (2017) 第 076301 号

责任编辑: 陈晓猛

印 刷: 三河市良远印务有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×980 1/16 印张: 37.75 字数: 720 千字

版 次: 2017 年 5 月第 1 版

印 次: 2017 年 5 月第 1 次印刷

定 价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方: 010-51260888-819, faq@phei.com.cn。

前言

这是一个数据大爆炸的时代，互联网成为了数据传播的主要载体。大数据处理平台在现代化的互联网公司进行商业决策、规划发展、市场拓展等方面扮演着越来越重要的角色。Kafka 作为大数据平台的重要组件之一，受到越来越多的设计人员和开发人员的青睐，Kafka 的社区也变得越来越活跃，Kafka 本身的架构设计、应用场景也得到了长足的发展。

Kafka 最开始由 LinkedIn 设计开发，并于 2011 年年初开源，2012 年 10 月成为 Apache 基金会的顶级项目。目前 Kafka 为越来越多的分布式大数据处理系统提供支持，其中也包括著名的 Apache Spark, LinkedIn、Netflix、Uber、Verizon、网易、美团等互联网公司也选择以 Kafka 为基础搭建其大数据处理平台或消息中间件系统。随着 Kafka 的应用场景越来越丰富，用户对 Kafka 的吞吐量、可扩展性、稳定性和可维护性等有了更多的期许，也有很多开发人员参与到 Kafka 的开发建议制定和代码提交中。在 Kafka 0.10.X 版本中出现了很多令人欣喜的新功能，本书深入剖析了 Kafka 0.10.X 的内部设计和实现细节。

本书以 Kafka 0.10.0 版本源码为基础，深入剖析了 Kafka 的各个模块的实现，包括 Kafka 的生产者客户端、消费者客户端、服务端的各个模块以及常用的管理脚本。笔者对 Kafka 设计的理解和经验分享也穿插在了剖析源码的过程中，希望读者能够通过本书理解 Kafka 的设计原理和源码实现，同时也学习到 Kafka 中优秀的设计思想以及 Java 和 Scala 的编程技巧和规范。

如何阅读本书

由于本书的篇幅限制，本书并没有详细介绍 Kafka 源码中涉及的所有基础知识，例如 Java NIO、J.U.C 包中工具类的使用、命令行参数解析器的使用等，为方便读者阅读，笔者仅介绍了一些必须且重要的基础知识。在开始源码分析之前，希望读者按照第 1 章的相关介绍完成 Kafka 源码环境的搭建，并了解 Kafka 的核心概念，这样也可以有更好的学习效果。

本书共五章，它们互相之间的联系并不是很强，读者可以从头开始阅读，也可以选择自己感兴趣的章节进行学习。

第1章是Kafka的快速入门，其中介绍了Kafka的背景、特性以及应用场景。之后介绍了笔者在实践中遇到的一个以Kafka为中心的案例，并分析了在此案例中选择使用Kafka的具体原因和Kafka起到的关键作用。最后介绍了Kafka中的核心概念和Kafka源码调试环境的搭建。

第2章介绍了生产者客户端的设计特点和实现细节，剖析了KafkaProducer拦截消息、序列化消息、路由消息等功能的源码实现，介绍了RecordAccumulator的结构和实现。最后剖析了KafkaProducer中Sender线程的源码。

第3章介绍了Kafka的消息传递保证语义并给出了相关的实践建议，还介绍了Consumer Group Rebalance操作各个版本方案的原理和弊端。最后详细剖析了KafkaConsumer相关组件的运行原理和实现细节。

第4章介绍了构成Kafka服务端的各个组件，依次分析了Kafka网络层、API层、日志存储、DelayedOperationPurgatory组件、Kafka的副本机制、KafkaController、GroupCoordinator、Kafka的身份认证与权限控制以及Kafka监控相关的实现。本章是Kafka的核心内容，涉及较多的设计细节和编程技巧，希望读者阅读之后有所收获。

第5章介绍了Kafka提供的多个脚本工具的使用以及具体实现原理，了解这些脚本可以帮助管理人员快速完成一些常见的管理、运维、测试功能。

如果读者在阅读本书的过程中，发现任何不妥之处，请将您宝贵的意见和建议发送到邮箱 xxxlxy2008@163.com，也欢迎读者朋友通过此邮箱与笔者进行交流。

致谢

感谢电子工业出版社博文视点的陈晓猛老师，是您的辛勤工作让本书的出版成为可能。同时还要感谢许多我不知道名字的幕后工作人员为本书付出的努力。

感谢张占龙、张亚森、杨威、刘克刚、刘思等朋友在百忙之中抽出时间对本书进行审阅和推荐。感谢林放、米秀明、星亮亮、王松洋、褚洪洋、曾天宁、葛彬、赵美凯、顾聪慧、孙向川、段鑫冬、彭海蛟、赵仁伟等同事，帮助我解决工作中的困难。

感谢冯玉玉、李成伟，是你们让写作的过程变得妙趣横生，是你们让我更加积极、自信，也是你们的鼓励让我完成了本书的写作。

最后, 特别感谢我的母亲大人, 谢谢您默默为我做出的牺牲和付出, 您是我永远的女神。

徐郡明

读者服务

轻松注册成为博文视点社区用户 (www.broadview.com.cn), 您即可享受以下服务:

- **提交勘误:** 您对书中内容的修改意见可在【提交勘误】处提交, 若被采纳, 将获赠博文视点社区积分 (在您购买电子书时, 积分可用来抵扣相应金额)。
- **与作者交流:** 在页面下方【读者评论】处留下您的疑问或观点, 与作者和其他读者一同学习交流。

页面入口: <http://www.broadview.com.cn/31345>

二维码:



专家推荐

《Apache Kafka 源码剖析》一书深入浅出地分析了 Kafka 的源代码，无论是刚接触 Kafka 的菜鸟，还是已经有多年 Kafka 使用经验的老鸟，这本书都能让你有所收获。

——搜狗高级研发工程师 张亚森

Kafka 是大数据平台中的关键部分之一。《Apache Kafka 源码剖析》全面细致地剖析了 Kafka 的运行原理和架构设计，在带领读者进入 Kafka 源码世界的同时，也分析了许多设计经验，是一本不可多得的好书。

——华为高级研发工程师 张占龙

在阅读《Apache Kafka 源码剖析》时，作者在每一章节中都会给我意外之惊喜。作者对 Kafka 源代码已有相当深刻的理解，此书代码分析过程逻辑清晰，详略得当，实属不易。

——网易游戏高级数据挖掘研究员 杨威

大型分布式系统犹如一个生命，系统中各个服务犹如骨骼，其中的数据犹如血液，而 Kafka 犹如经络，串联整个系统。《Apache Kafka 源码剖析》通过大量的设计图展示、代码分析、示例分享，把 Kafka 的实现脉络展示在读者面前，帮助读者更好地研读 Kafka 代码。

——今日头条高级研发工程师 刘克刚

《Apache Kafka 源码剖析》中汇集了作者多年 Kafka 开发经验，为读者深入学习 Kafka 实现指明了方向。对于想学习 Kafka 的程序员来说，这是一本非常不错的进阶书籍。

——美团高级研发工程师 刘思

目录

第 1 章 快速入门	1
1.1 Kafka 简介	1
1.2 以 Kafka 为中心的解决方案	2
1.3 Kafka 核心概念	6
1.4 搭建 Kafka 源码环境	16
本章小结	26
第 2 章 生产者	27
2.1 KafkaProducer 使用示例	27
2.2 KafkaProducer 分析	30
2.2.1 ProducerInterceptors&ProducerInterceptor	36
2.2.2 Kafka 集群元数据	37
2.2.3 Serializer&Deserializer	42
2.2.4 Partitioner	43
2.3 RecordAccumulator 分析	45
2.3.1 MemoryRecords	46
2.3.2 RecordBatch	49
2.3.3 BufferPool	53
2.3.4 RecordAccumulator	57
2.4 Sender 分析	65

2.4.1 创建请求	67
2.4.2 KSelector	70
2.4.3 InFlightRequests	76
2.4.4 MetadataUpdater	77
2.4.5 NetworkClient	83
本章小结	90
第3章 消费者	91
3.1 KafkaConsumer 使用示例	91
3.2 传递保证语义 (Delivery guarantee semantic)	93
3.3 Consumer Group Rebalance 设计	96
3.4 KafkaConsumer 分析	100
3.4.1 ConsumerNetworkClient	101
3.4.2 SubscriptionState	109
3.4.3 ConsumerCoordinator	114
3.4.4 PartitionAssignor 分析	117
3.4.5 Heartbeat 分析	119
3.4.6 Rebalance 实现	126
3.4.7 offset 操作	143
3.4.8 Fetcher	150
3.4.9 KafkaConsumer 分析总结	160
本章小结	164
第4章 Kafka 服务端	165
4.1 网络层	166
4.1.1 Reactor 模式	166
4.1.2 SocketServer	169
4.1.3 AbstractServerThread	172
4.1.4 Acceptor	174

4.1.5 Processor	177
4.1.6 RequestChannel	183
4.2 API 层	187
4.2.1 KafkaRequestHandler	188
4.2.2 KafkaApis	190
4.3 日志存储	191
4.3.1 基本概念	191
4.3.2 FileMessageSet	192
4.3.3 ByteBufferMessageSet	198
4.3.4 OffsetIndex	212
4.3.5 LogSegment	215
4.3.6 Log	220
4.3.7 LogManager	233
4.4 DelayedOperationPurgatory 组件	260
4.4.1 TimingWheel	260
4.4.2 SystemTimer	265
4.4.3 DelayedOperation	267
4.4.4 DelayedOperationPurgatory	269
4.4.5 DelayedProduce	273
4.4.6 DelayedFetch	281
4.5 副本机制	290
4.5.1 副本	291
4.5.2 分区	293
4.5.3 ReplicaManager	304
4.6 KafkaController	339
4.6.1 ControllerChannelManager	342
4.6.2 ControllerContext	345
4.6.3 ControllerBrokerRequestBatch	347
4.6.4 PartitionStateMachine	351
4.6.5 PartitionLeaderSelector	360

4.6.6	ReplicaStateMachine	363
4.6.7	ZooKeeper Listener	369
4.6.8	KafkaController 初始化与故障转移	397
4.6.9	处理 ControlledShutdownRequest	406
4.7	GroupCoordinator	409
4.7.1	GroupMetadataManager	412
4.7.2	GroupCoordinator 分析	432
4.8	身份认证与权限控制	460
4.8.1	配置 SASL/PLAIN 认证	461
4.8.2	身份认证	464
4.8.3	权限控制	491
4.9	Kafka 监控	500
4.9.1	JMX 简介	501
4.9.2	Metrics 简介	506
4.9.3	Kafka 中的 Metrics	512
4.9.4	Kafka 的监控功能	521
4.9.5	监控 KSelector 的指标	534
第5章	Kafka Tool	543
5.1	kafka-server-start 脚本	544
5.2	kafka-topics 脚本	547
5.2.1	创建 Topic	548
5.2.2	修改 Topic	555
5.3	kafka-preferred-replica-election 脚本	558
5.4	kafka-reassign-partitions 脚本	560
5.5	kafka-console-producer 脚本	565
5.6	kafka-console-consumer 脚本	566
5.7	kafka-consumer-groups 脚本	569

5.8 DumpLogSegments	573
5.9 kafka-producer-perf-test 脚本	577
5.10 kafka-consumer-perf-test 脚本	581
5.11 kafka-mirror-maker 脚本	583
本章小结	591

第 1 章

快速入门

1.1 Kafka 简介

Apache Kafka 是一种分布式的、基于发布 / 订阅的消息系统，由 Scala 语言编写而成。它具备快速、可扩展、可持久化的特点。Kafka 最初由 LinkedIn 开发，并于 2011 年初开源，2012 年 10 月从 Apache 孵化器毕业，成为 Apache 基金会的顶级项目。目前，越来越多的开源分布式处理系统支持与 Kafka 集成，例如：Apache Storm、Spark。也有越来越多的公司在 Kafka 的基础上建立了近乎实时的信息处理平台，例如：LinkedIn、Netflix、Uber 和 Verizon。在国内也有很多互联网公司在生产环境中使用 Kafka 作为其消息中间件。

Kafka 之所以受到越来越多开发人员的青睐，主要与其关键特性相关。

- Kafka 具有近乎实时性的消息处理能力，即使面对海量消息也能够高效地存储消息和查询消息。Kafka 将消息保存在磁盘中，在其设计理念中并不惧怕磁盘操作，它以顺序读写的方式访问磁盘，从而避免了随机读写磁盘导致的性能瓶颈。
- Kafka 支持批量读写消息，并且会对消息进行批量压缩，这样既提高了网络的利用率，也提高了压缩效率。
- Kafka 支持消息分区，每个分区中的消息保证顺序传输，而分区之间则可以并发操作，这样就提高了 Kafka 的并发能力。
- Kafka 也支持在线增加分区，支持在线水平扩展。
- Kafka 支持为每个分区创建多个副本，其中只会有一个 Leader 副本负责读写，其

他副本只负责与 Leader 副本进行同步，这种方式提高了数据的容灾能力。Kafka 会将 Leader 副本均匀地分布在集群中的服务器上，实现性能最大化。

随着 Kafka 在各大公司的实践应用，Kafka 的应用场景也变得越来越丰富。

- 在应用系统中可以将 Kafka 作为传统的消息中间件，实现消息队列和消息的发布 / 订阅，在某些场景下，性能会超越 RabbitMQ、ActiveMQ 等传统的消息中间件。
- Kafka 也被用作系统中的数据总线，将其接入多个子系统中，子系统会将产生的数据发送到 Kafka 中保存，之后流转到的系统中。
- Kafka 还可以用作日志收集中心，多个系统产生的日志统一收集到 Kafka 中，然后由数据分析平台进行统一处理。日志会被 Kafka 持久化到磁盘，所以同时支持离线数据分析和实时数据处理。
- 现在也有开发人员基于 Kafka 设计数据库主从同步的工具。

1.2 以 Kafka 为中心的解决方案

在大数据、高并发的系统中，为了突破瓶颈，会将系统进行水平扩展和垂直拆分，形成独立的服务。每个独立的服务背后，可能是一个集群在对外提供服务。这就会碰到的问题，整个系统是由多个服务（子系统）组成的，数据需要在各个服务中不停流转。如果数据在各个子系统中传输时，速度过慢，就会形成瓶颈，降低整个系统的性能。

下面介绍的场景是笔者工作中遇到的一个案例，在一个政企信息化的云平台网站上，用户与网站交互的很多操作行为（例如，浏览某些新闻等）都会被记录下来，等待后台的多个子系统进行消费，其中比较重要的几个子系统是利用这些数据进行机器学习或是数据挖掘，产生用户的侧写。这样，网站就可以根据用户的侧写，推送给他们需要的配置和查询信息。图 1-1 就是这个云平台系统的架构图，其中每一个箭头都表示一条数据流。

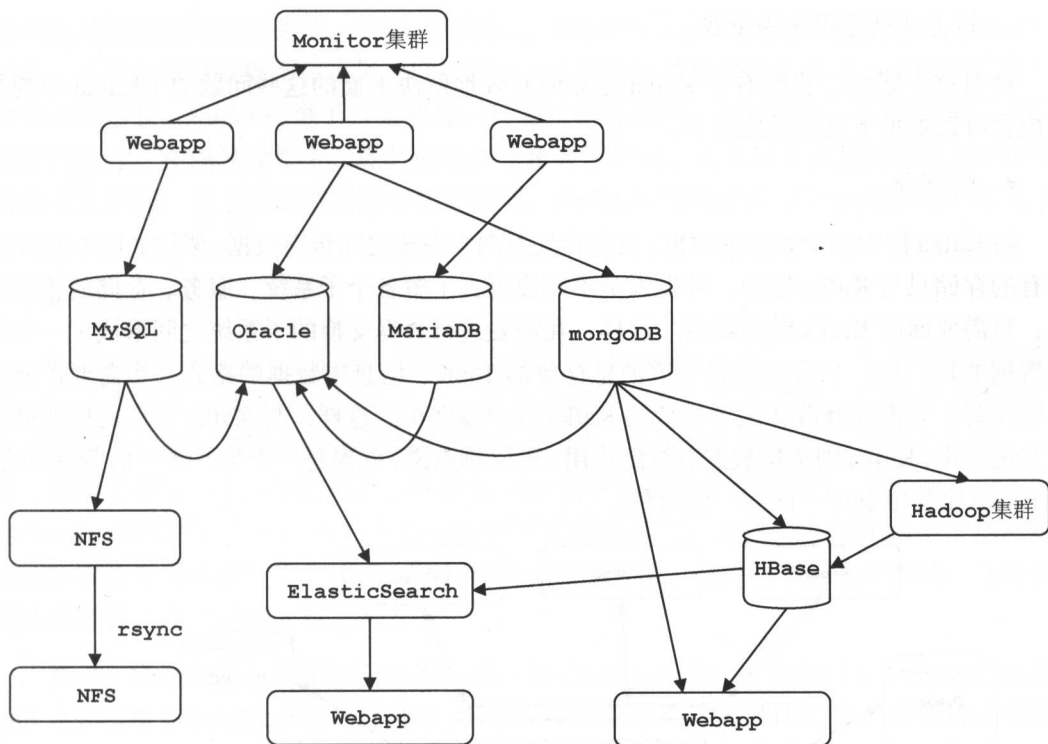


图 1-1

上面的架构中涉及的子系统、存储、服务种类繁多，而且它们之间都存在较强的耦合，会出现下面的问题：

- 由于子系统之间存在的耦合性，两个存储之间要进行数据交换的话，开发人员就必须了解这两个存储系统的 API，不仅是开发成本，就连维护成本也会很高。一旦其中一个子系统发生变化，就可能影响其他多个子系统，这简直就是一场灾难。
- 在某些应用场景中，数据的顺序性尤为重要，一旦数据出现乱序，就会影响最终的计算结果，降低用户体验，这就提高了开发的难度。
- 除了考虑数据顺序性的要求，还要考虑数据重传等提高可靠性的机制，毕竟通过网络进行传输并不可靠，可能出现丢失数据的情况。
- 进行数据交换的两个子系统，无论哪一方宕机，重新上线之后，都应该恢复到之前的传输位置，继续传输。尤其是对于非幂等性的操作，恢复到错误的传输位置，就会导致错误的结果。
- 随着业务量的增长，系统之间交换的数据量会不断地增长，水平可扩展的数据传

输方式就显得尤为重要。

针对这个案例，我们看看 Kafka 是如何有效地解决上面的这些问题的（Kafka 中的相关概念可以参见下文相关内容）。

- 解耦合

将 Kafka 作为整个系统的中枢，负责在任意两个系统之间传递数据。架构如图 1-2 所示，所有的存储只与 Kafka 通信，开发人员不需要再去了解各个子系统、服务、存储的相关接口，只需要面向 Kafka 编程即可。这样，在需要进行数据交换的子系统之间形成了一个基于数据的接口层，只有这两者知道消息存放的 Topic、消息中数据的格式。当需要扩展消息格式时，只需要修改相关子系统的 Kafka 客户端即可。这样，与 Kafka 通信的模块就可以实现复用，Kafka 则承担数据总线的作用。更简单点说，就像是一个生产者—消费者模式，而 Kafka 则扮演其中“队列”的角色。

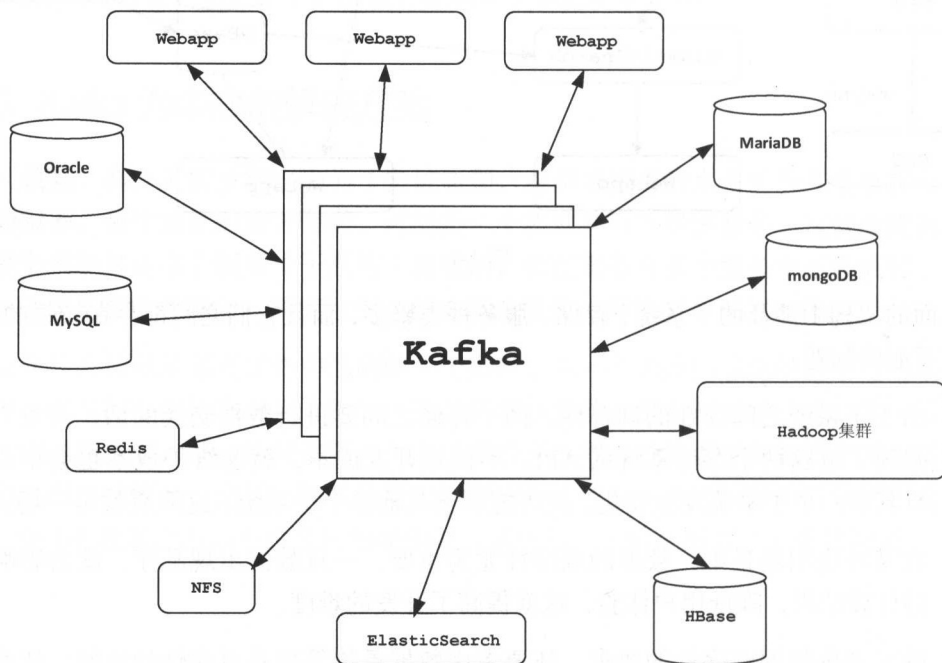


图 1-2

- 数据持久化

在分布式系统中，各个组件是通过网路连接起来的。一般认为网络传输是不可靠的，当数据在两个组件之间进行传递的时候，传输过程可能会失败。除非数据被持久化到磁盘，

否则就可能造成消息的丢失。Kafka 把数据以消息的形式持久化到磁盘，即使 Kafka 出现宕机，也可以保证数据不会丢失，通过这一方式规避了数据丢失风险。为了避免磁盘上的数据不断增长，Kafka 提供了日志清理、日志压缩等功能，对过时的、已经处理完成的数据进行清除。在磁盘操作中，耗时最长的就是寻道时间，这是导致磁盘的随机 I/O 性能很差的主要原因。为了提高消息持久化的性能，Kafka 采用顺序读写的方式访问，实现了高吞吐量。

- 扩展与容灾

Kafka 的每个 Topic（主题）都可以分为多个 Partition（分区），每个分区都有多个 Replica（副本），实现消息冗余备份。每个分区中的消息是不同的，这类似于数据库中水平切分的思想，提高了并发读写的能力。而同一分区的不同副本中保存的是相同的消息，副本之间是一主多从的关系，其中 Leader 副本负责处理读写请求，Follower 副本则只与 Leader 副本进行消息同步，当 Leader 副本出现故障时，则从 Follower 副本中重新选举 Leader 副本对外提供服务。这样，通过提高分区的数量，就可以实现水平扩展；通过提高副本的数量，就可以提高容灾能力。

Kafka 的容灾能力不仅体现在服务端，在 Consumer 端也有相关设计。Consumer 使用 pull 方式从服务端拉取消息，并且在 Consumer 端保存消费的具体位置，当消费者宕机后恢复上线，可以根据自己保存的消费位置重新拉取需要的消息进行消费，这就不会造成消息丢失。也就是说，Kafka 不决定何时、如何消费消息，而是 Consumer 自己决定何时、如何消费消息。

Kafka 还支持 Consumer 的水平扩展能力。我们可以让多个 Consumer 加入一个 Consumer Group（消费组），在一个 Consumer Group 中，每个分区只能分配给一个 Consumer 消费，当 Kafka 服务端通过增加分区数量进行水平扩展后，我们可以向 Consumer Group 中增加新的 Consumer 来提高整个 Consumer Group 的消费能力。当 Consumer Group 中的一个 Consumer 出现故障下线时，会通过 Rebalance 操作将下线 Consumer 负责处理的分区分配给其他 Consumer 继续处理；当下线 Consumer 重新上线加入 Consumer Group 时，会再进行一次 Rebalance 操作，重新分配分区。当然，一个 Consumer Group 可以订阅很多不同的 Topic，每个 Consumer 可以同时处理多个分区。

- 顺序保证

在很多场景下，数据处理的顺序都很重要，不同的顺序就可能导致不同的计算结果。Kafka 保证一个 Partition 内消息的有序性，但是并不保证多个 partition 之间的数据有顺序。

- 缓冲 & 峰值处理能力

在访问量剧增的情况下,应用仍然需要继续发挥作用,但是这样的突发流量并不常见。如图 1-3 所示,在 9 点到 10 点之间,是此云平台系统的访问峰值,而其他时间的访问量则很少。

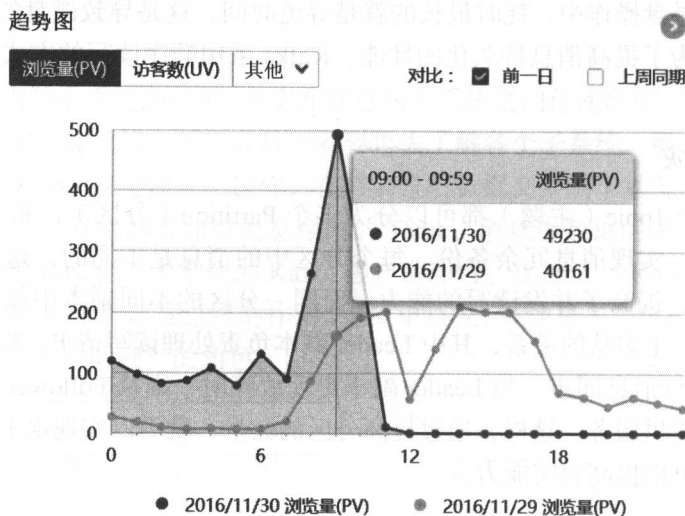


图 1-3

如果按照处理这类峰值请求量为标准来投入资源的话,会有相当一部分资源处于待命状态,这无疑是巨大的浪费。使用 Kafka 能够使关键组件顶住突发的访问压力,而不会因为突发的峰值请求而使系统完全崩溃不可用。

• 异步通信

Kafka 为系统提供了异步处理能力。例如,两个系统需要通过网络进行数据交换,其中一端可以把一个消息放入 Kafka 中后立即返回继续执行其他路基,不需要等待对端的响应。待后者将处理结果放入 Kafka 中之后,前者可以从其中获取并解析响应。

1.3 Kafka 核心概念

如果读者已经对下面的概念非常熟悉,可以快速阅读或直接跳过本节,直接开始搭建 Kafka 的源码调试环境。

• 消息

消息是 Kafka 中最基本的数据单元。消息由一串字节构成，其中主要由 key 和 value 构成，key 和 value 也都是 byte 数组。key 的主要作用是根据一定的策略，将此消息路由到指定的分区中，这样就可以保证包含同一 key 的消息全部写入同一分区中，key 可以是 null。消息的真正有效负载是 value 部分的数据。为了提高网络和存储的利用率，生产者会批量发送消息到 Kafka，并在发送之前对消息进行压缩，具体的细节在本书后面的章节会详细介绍。

- Topic & 分区 & Log

Topic 是用于存储消息的逻辑概念，可以看作一个消息集合。每个 Topic 可以有多个生产者向其中推送 (push) 消息，也可以有任意多个消费者消费其中的消息，如图 1-4 所示。

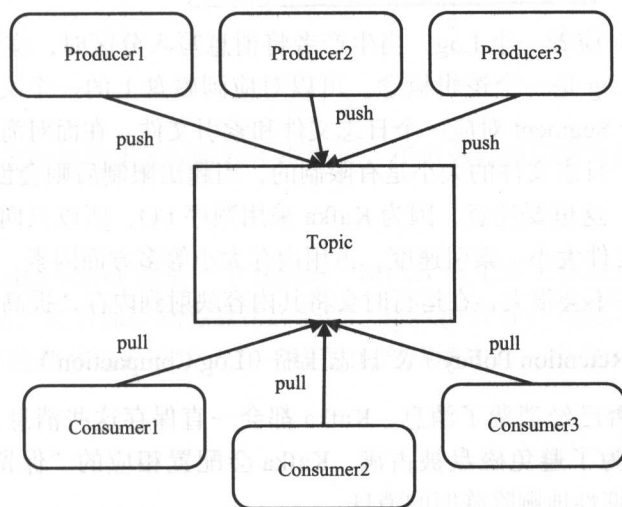


图 1-4

每个 Topic 可以划分成多个分区（每个 Topic 都至少有一个分区），同一 Topic 下的不同分区包含的消息是不同的。每个消息在被添加到分区时，都会被分配一个 offset，它是消息在此分区中的唯一编号，Kafka 通过 offset 保证消息在分区内的顺序，offset 的顺序性不跨分区，即 Kafka 只保证在同一个分区内的消息是有序的；同一 Topic 的多个分区内的消息，Kafka 并不保证其顺序性，如图 1-5 所示。

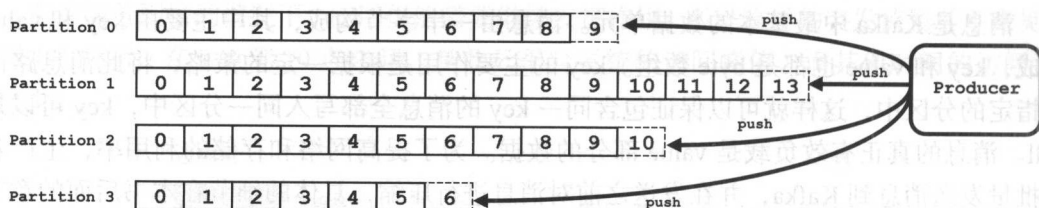


图 1-5

同一 Topic 的不同分区会分配在不同的 Broker (Broker 的概念见下文) 上。分区是 Kafka 水平扩展性的基础，我们可以通过增加服务器并在其上分配 Partition 的方式来增加 Kafka 的并行处理能力。

分区在逻辑上对应着一个 Log，当生产者将消息写入分区时，实际上是写入到了分区对应的 Log 中。Log 是一个逻辑概念，可以对应到磁盘上的一个文件夹。Log 由多个 Segment 组成，每个 Segment 对应一个日志文件和索引文件。在面对海量数据时，为避免出现超大文件，每个日志文件的大小是有限制的，当超出限制后则会创建新的 Segment，继续对外提供服务。这里要注意，因为 Kafka 采用顺序 I/O，所以只向最新的 Segment 追加数据。为了权衡文件大小、索引速度、占用内存大小等多方面因素，索引文件采用稀疏索引的方式，大小并不会很大，在运行时会将内容映射到内存，提高索引速度。

• 保留策略 (Retention Policy) & 日志压缩 (Log Compaction)

无论消费者是否已经消费了消息，Kafka 都会一直保存这些消息，但并不会像数据库那样长期保存。为了避免磁盘被占满，Kafka 会配置相应的“保留策略” (retention policy)，以实现周期性地删除陈旧的消息。

Kafka 中有两种“保留策略”：一种是根据消息保留的时间，当消息在 Kafka 中保存的时间超过了指定时间，就可以被删除；另一种是根据 Topic 存储的数据大小，当 Topic 所占的日志文件大小大于一个阈值，则可以开始删除最旧的消息。Kafka 会启动一个后台线程，定期检查是否存在可以删除的消息。“保留策略”的配置是非常灵活的，可以有全局的配置，也可以针对 Topic 进行配置覆盖全局配置。

除此之外，Kafka 还会进行“日志压缩” (Log Compaction)。在很多场景中，消息的 key 与 value 的值之间的对应关系是不断变化的，就像数据库中的数据会不断被修改一样，消费者只关心 key 对应的最新 value 值。此时，可以开启 Kafka 的日志压缩功能，Kafka 会在后台启动一个线程，定期将相同 key 的消息进行合并，只保留最新的 value 值。日志压缩的工作原理如图 1-6 所示，图 1-6 展示了一次日志压缩过程的简化版本，为了图片清晰，只展示了 key3 的压缩过程。

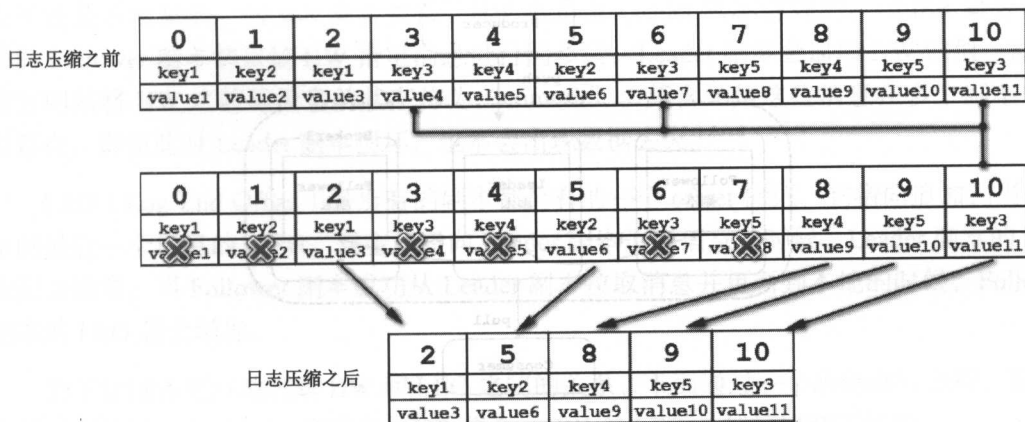


图 1-6

保留策略和日志压缩的相关配置和实现细节，在本书后面的章节会详细介绍。

• Broker

一个单独的 Kafka server 就是一个 Broker。Broker 的主要工作就是接收生产者发过来的消息，分配 offset，之后保存到磁盘中；同时，接收消费者、其他 Broker 的请求，根据请求类型进行相应处理并返回响应。在一般的生产环境中，一个 Broker 独占一台物理服务器。

• 副本

Kafka 对消息进行了冗余备份，每个 Partition 可以有多个副本，每个副本中包含的消息是一样的（在同一时刻，副本之间其实并不是完全一样的，本书后面在介绍副本机制的时候会再进行说明）。每个分区至少有一个副本，当分区中只有一个副本时，就只有 Leader 副本，没有 Follower 副本。

每个分区的副本集合中，都会选举出一个副本作为 Leader 副本，Kafka 在不同的场景下会采用不同的选举策略，具体策略和原理在后面详细介绍。所有的读写请求都由选举出的 Leader 副本处理，其他都作为 Follower 副本，Follower 副本仅仅是从 Leader 副本处把数据拉取到本地之后，同步更新到自己的 Log 中。图 1-7 展示了一个拥有三个 Replica 的 Partition。

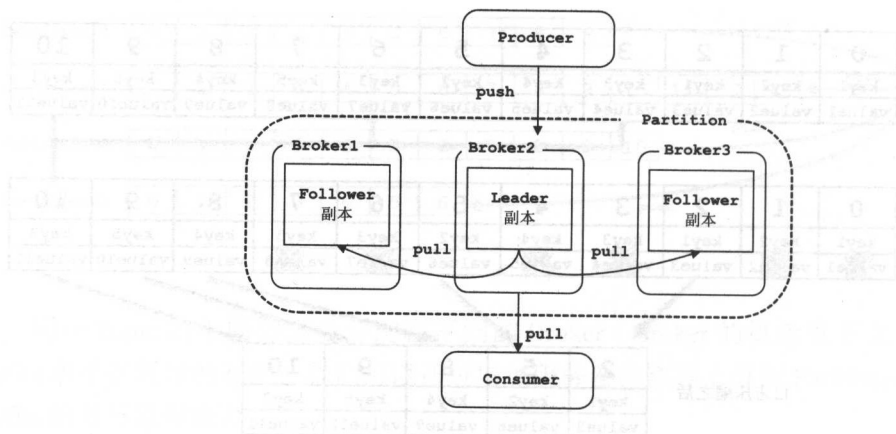


图 1-7

一般情况下，同一分区的多个副本会被分配到不同的 Broker 上，这样，当 Leader 所在的 Broker 宕机之后，可以重新选举新的 Leader，继续对外提供服务。

• ISR 集合

ISR (In-Sync Replica) 集合表示的是目前“可用” (alive) 且消息量与 Leader 相差不多的副本集合，这是整个副本集合的一个子集。“可用”和“相差不多”都是很模糊的描述，其实际含义是 ISR 集合中的副本必须满足下面两个条件：

(1) 副本所在节点必须维持着与 ZooKeeper 的连接。

(2) 副本最后一条消息的 offset 与 Leader 副本的最后一条消息的 offset 之间的差值不能超出指定的阈值。

每个分区中的 Leader 副本都会维护此分区的 ISR 集合。写请求首先由 Leader 副本处理，之后 Follower 副本会从 Leader 上拉取写入的消息，这个过程会有一定的延迟，导致 Follower 副本中保存的消息略少于 Leader 副本，只要未超出阈值都是可以容忍的。如果一个 Follower 副本出现异常，比如：宕机，发生长时间 GC 而导致 Kafka 僵死或是网络断开连接导致长时间没有拉取消息进行同步，就会违反上面的两个条件，从而被 Leader 副本踢出 ISR 集合。当 Follower 副本从异常中恢复之后，会继续与 Leader 副本进行同步，当 Follower 副本“追上” (即最后一条消息的 offset 的差值小于指定阈值) Leader 副本的时候，此 Follower 副本会被 Leader 副本重新加入到 ISR 中。

• HW&LEO

HW (HighWatermark) 和 LEO 与上面的 ISR 集合紧密相关。HW 标记了一个特殊的 offset，当消费者处理消息的时候，只能拉取到 HW 之前的消息，HW 之后的消息对消费

者来说是不可见的。与 ISR 集合类似，HW 也是由 Leader 副本管理的。当 ISR 集合中全部的 Follower 副本都拉取 HW 指定消息进行同步后，Leader 副本会递增 HW 的值。Kafka 官方网站将 HW 之前的消息的状态称为“commit”，其含义是这些消息在多个副本中同时存在，即使此时 Leader 副本损坏，也不会出现数据丢失。

LEO (Log End Offset) 是所有的副本都会有一个 offset 标记，它指向追加到当前副本的最后一个消息的 offset。当生产者向 Leader 副本追加消息的时候，Leader 副本的 LEO 标记会递增；当 Follower 副本成功从 Leader 副本拉取消息并更新到本地的時候，Follower 副本的 LEO 就会增加。

为了让读者更好地理解 HW 和 LEO 之间的关系，下面通过一个示例进行分析，图 1-8 中展示了针对 offset 为 11 的消息，ISR 集合、HW 与 LEO 是如何协调工作的：

- ① Producer 向此 Partition 推送消息。
 - ② Leader 副本将消息追加到 Log 中，并递增其 LEO。
 - ③ Follower 副本从 Leader 副本拉取消息进行同步。
 - ④ Follower 副本将拉取到的消息更新到本地 Log 中，并递增其 LEO。
 - ⑤当 ISR 集合中所有副本都完成了对 offset=11 的消息的同步，Leader 副本会递增 HW。
- HW。

在①~⑤步完成之后，offset=11 的消息就对生产者可见了。

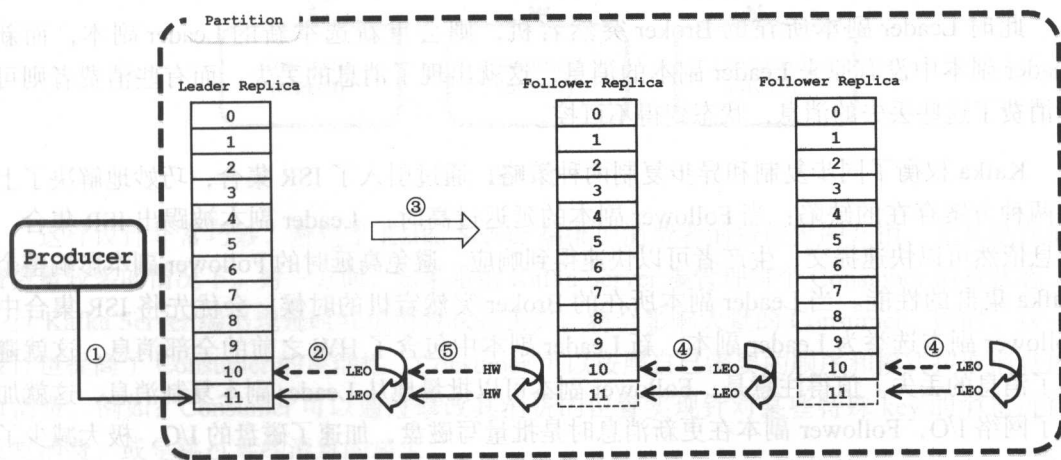


图 1-8

了解了 Replica 复制的原理之后，请读者考虑一下，为什么 Kafka 要这么设计？在分布式存储中，冗余备份是常见的一种设计，常用的方案有同步复制和异步复制：

- 同步复制要求所有能工作的 Follower 副本都复制完，这条消息才会被认为提交成功。一旦有一个 Follower 副本出现故障，就会导致 HW 无法完成递增，消息就无法提交，生产者获取不到消息。这种情况下，故障的 Follower 副本会拖慢整个系统的性能，甚至导致整个系统不可用。
- 异步复制中，Leader 副本收到生产者推送的消息后，就认为此消息提交成功。Follower 副本则异步地从 Leader 副本同步消息。这种设计虽然避免了同步复制的问题，但同样也存在一定的风险。现在假设所有 Follower 副本的同步速度都比较慢，它们保存的消息量都远远落后于 Leader 副本，如图 1-9 所示。

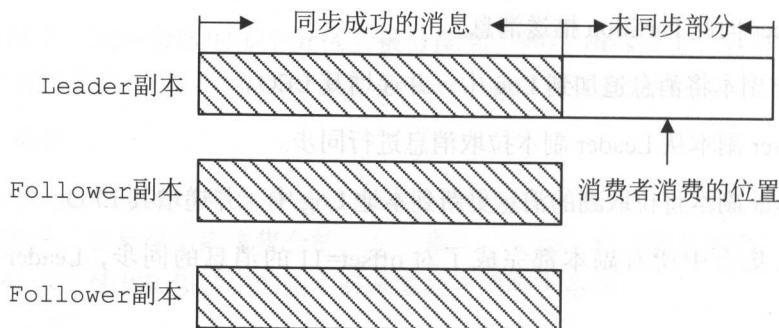


图 1-9

此时 Leader 副本所在的 Broker 突然宕机，则会重新选举新的 Leader 副本，而新 Leader 副本中没有原来 Leader 副本的消息，这就出现了消息的丢失，而有些消费者则可能消费了这些丢失的消息，状态变得不可控。

Kafka 权衡了同步复制和异步复制两种策略，通过引入了 ISR 集合，巧妙地解决了上面两种方案存在的缺陷：当 Follower 副本的延迟过高时，Leader 副本被踢出 ISR 集合，消息依然可以快速提交，生产者可以快速得到响应，避免高延时的 Follower 副本影响整个 Kafka 集群的性能。当 Leader 副本所在的 Broker 突然宕机的时候，会优先将 ISR 集合中 Follower 副本选举为 Leader 副本，新 Leader 副本中包含了 HW 之前的全部消息，这就避免了消息的丢失。值得注意的是，Follower 副本可以批量地从 Leader 副本复制消息，这就加快了网络 I/O，Follower 副本在更新消息时是批量写磁盘，加速了磁盘的 I/O，极大减少了 Follower 与 Leader 的差距。

- Cluster&Controller

多个 Broker 可以做成一个 Cluster（集群）对外提供服务，每个 Cluster 当中会选举出一个 Broker 来担任 Controller，Controller 是 Kafka 集群的指挥中心，而其他 Broker 则听从 Controller 指挥实现相应的功能。Controller 负责管理分区的状态、管理每个分区的副本状态、监听 Zookeeper 中数据的变化等工作。Controller 也是一主多从的实现，所有 Broker 都会监听 Controller Leader 的状态，当 Leader Controller 出现故障时则重新选举新的 Controller Leader。Controller 的具体细节在后面介绍。

- 生产者

生产者（Producer）的主要工作是生产消息，并将消息按照一定的规则推送到 Topic 的分区中。这里选择分区的“规则”可以有很多种，例如：根据消息的 key 的 Hash 值选择分区，或按序轮训全部分区的方式。

- 消费者

消费者（Consumer）的主要工作是从 Topic 中拉取消息，并对消息进行消费。某个消费者消费到 Partition 的哪个位置（offset）的相关信息，是 Consumer 自己维护的。在图 1-10 中，三个消费者同时消费同一个分区，各自管理自己的消费位置。

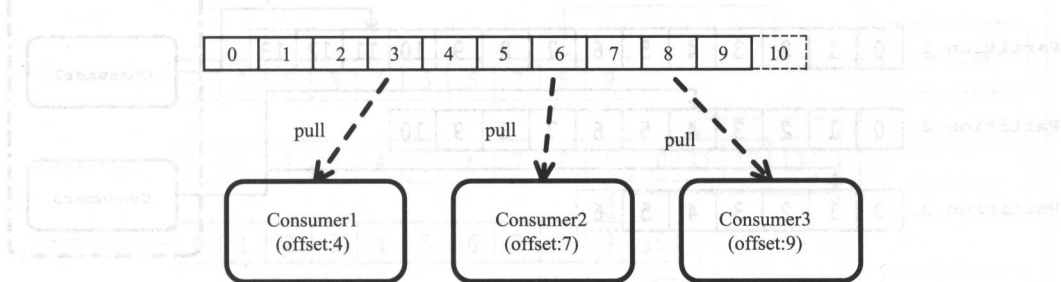


图 1-10

这样设计非常巧妙，避免了 Kafka Server 端维护消费者消费位置的开销，尤其是在消费数量较多的情况下。另一方面，如果是由 Kafka Server 端管理每个 Consumer 消费状态，一旦 Kafka Server 端出现延时或是消费状态丢失，将会影响大量的 Consumer。同时，这一设计也提高了 Consumer 的灵活性，Consumer 可以按照自己需要的顺序和模式拉取消息进行消费。例如：Consumer 可以通过修改其消费的位置实现针对某些特殊 key 的消息进行反复消费，或是跳过某些消息的需求。

• Consumer Group

在 Kafka 中，多个 Consumer 可以组成一个 Consumer Group，一个 Consumer 只能属于一个 Consumer Group。Consumer Group 保证其订阅的 Topic 的每个分区只被分配给此 Consumer Group 中的一个消费者处理。如果不同 Consumer Group 订阅了同一 Topic，Consumer Group 彼此之间不会干扰。这样，如果要实现一个消息可以被多个消费者同时消费（“广播”）的效果，则将每个消费者放入单独的一个 Consumer Group；如果要实现一个消息只被一个消费者消费（“独占”）的效果，则将所有的 Consumer 放入一个 Consumer Group 中。在 Kafka 官网的介绍中，将 Consumer Group 称为“逻辑上的订阅者”（logical subscriber），从这个角度看，是有一定道理的。

图 1-11 展示了一个 Consumer Group 中消费者与分区之间的对应关系，其中，Consumer1 和 Consumer2 分别消费 Partition0 和 Partition1，而 Partition2 和 Partition3 同时分配给了 Consumer3 进行处理。

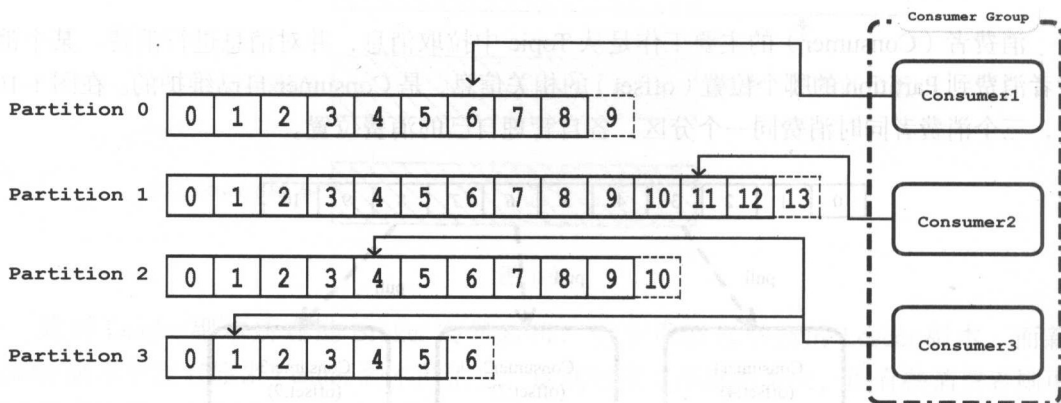


图 1-11

Consumer Group 除了实现“独占”和“广播”模式的消息处理，Kafka 还通过 Consumer Group 实现了消费者的水平扩展和故障转移。在图 1-11 中，当 Consumer3 的处理能力不足以处理两个 Partition 中的数据时，可以通过向 Consumer Group 中添加消费者的方式，触发 Rebalance 操作重新分配分区与消费者的对应关系，从而实现水平扩展。如图 1-12 所示，添加 Consumer4 之后，Consumer3 只消费 Partition3 中的消息，Partition4 中的消息则由 Consumer4 来消费。

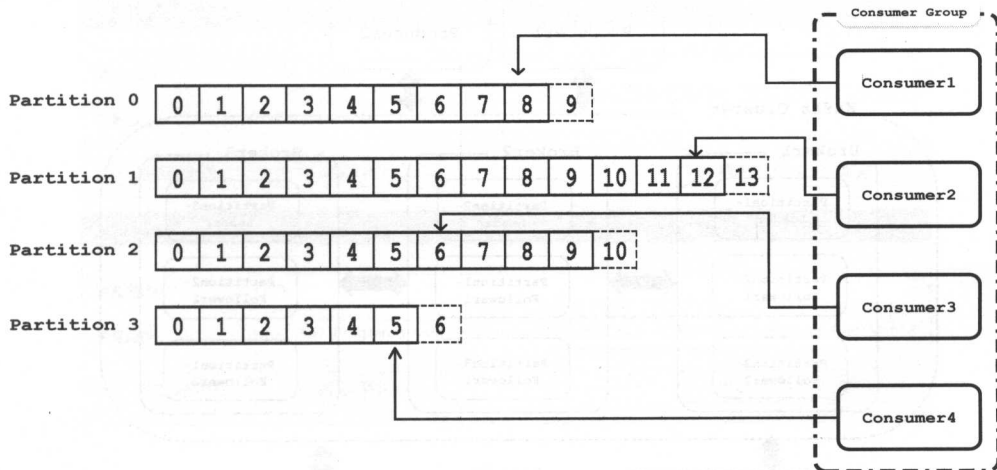


图 1-12

下面来看消费者出现故障的场景，当 Consumer4 宕机时，Consumer Group 会自动重新分配分区，如图 1-13 所示，由 Consumer3 接管 Consumer4 对应的分区继续处理。

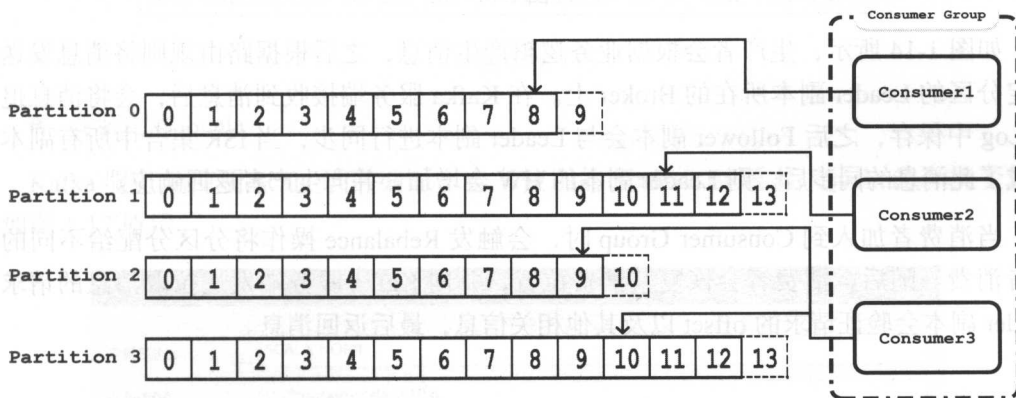


图 1-13

注意，Consumer Group 中消费者的数量并不是越多越好，当其中消费者数量超过分区的数量时，会导致有消费者分配不到分区，从而造成消费者的浪费。

介绍完 Kafka 的核心概念，我们通过图 1-14 进行总结，并从更高的视角审视整个 Kafka 集群的架构。

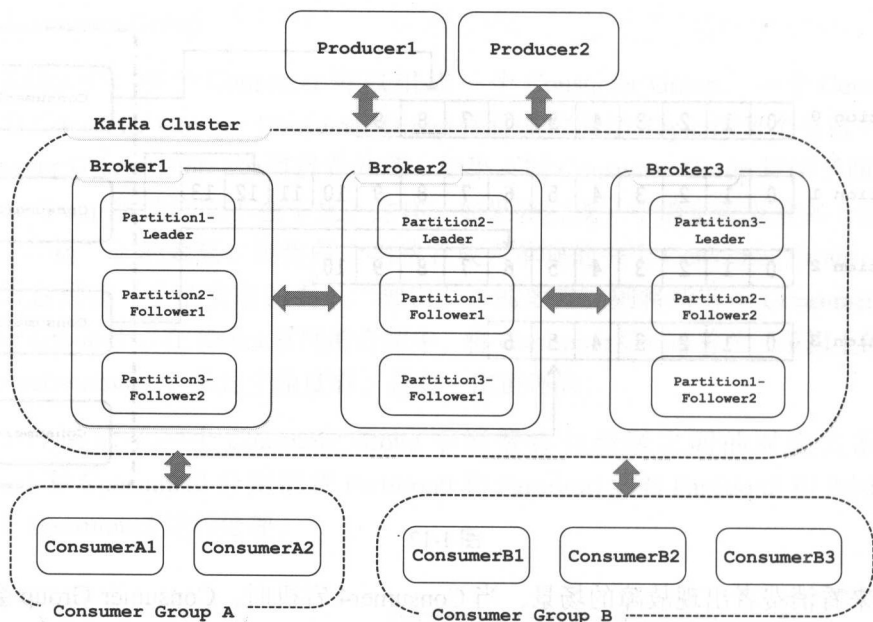


图 1-14

如图 1-14 所示，生产者会根据业务逻辑产生消息，之后根据路由规则将消息发送到指定分区的 Leader 副本所在的 Broker 上。在 Kafka 服务端接收到消息后，会将消息追加到 Log 中保存，之后 Follower 副本会与 Leader 副本进行同步，当 ISR 集合中所有副本都完成了此消息的同步后，则 Leader 副本的 HW 会增加，并向生产者返回响应。

当消费者加入到 Consumer Group 时，会触发 Rebalance 操作将分区分配给不同的消费者消费。随后，消费者会恢复其消费位置，并向 Kafka 服务端发送拉取消息的请求，Leader 副本会验证请求的 offset 以及其他相关信息，最后返回消息。

1.4 搭建 Kafka 源码环境

在开始分析 Kafka 的源码之前，我们先要动手搭建 Kafka 源码的调试环境。需要准备的软件有：Java JDK、Scala-2.10、gradle-3.1、ZooKeeper-3.4.9。本书选择的 Kafka 版本是 kafka-0.10.0.1，IDE 环境是 IntelliJ IDEA。

- 安装配置 JDK

本书是在 Window 系统上搭建 Kafka 的源码环境，使用的 JDK 版本是 1.8.0，环境变量配置如图 1-15 所示。JDK 安装以及环境变量的配置不再赘述，请读者自行查阅相关文档。

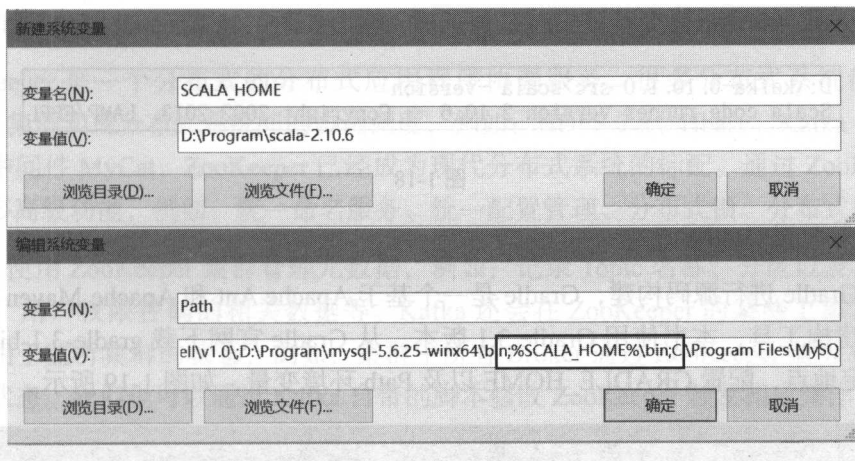


图 1-15

使用 `java -version` 命令验证，输出如图 1-16 所示。

```
d:\>java -version
java version "1.8.0_91"
Java(TM) SE Runtime Environment (build 1.8.0_91-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.91-b15, mixed mode)
```

图 1-16

• 安装配置 Scala

Kafka 服务端使用 Scala 语言编写，本书使用的 Scala-2.10 版本，Scala 的环境变量配置如图 1-17 所示。

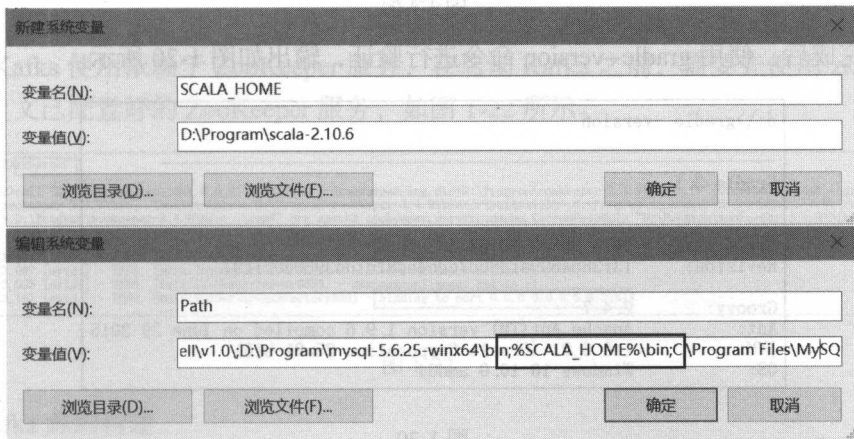


图 1-17

使用 `scala -version` 命令验证安装是否正确, 输出如图 1-18 所示。

```
D:\kafka-0.10.1.0-src>scala -version
Scala code runner version 2.10.6 -- Copyright 2002-2013, LAMP/EPFL
```

图 1-18

• 安装配置 Gradle

使用 Gradle 进行源码构建, Gradle 是一个基于 Apache Ant 和 Apache Maven 概念的项目自动化建构工具, 本书使用 Gradle 3.1 版本。从 Gradle 官网下载 `gradle-3.1-bin.zip`, 解压缩到指定地点, 配置 `GRADLE_HOME` 以及 Path 环境变量, 如图 1-19 所示。

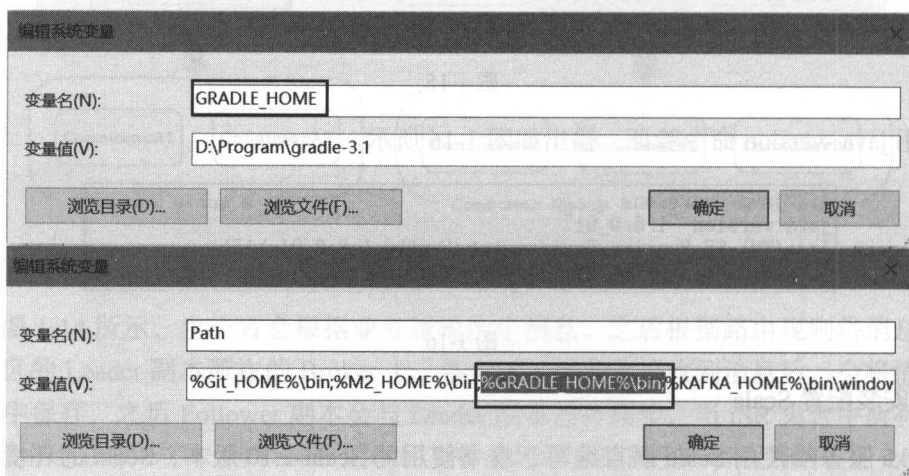


图 1-19

配置完成后, 使用 `gradle -version` 命令进行验证, 输出如图 1-20 所示。

```
d:\>gradle -version

-----
Gradle 3.1
-----

Build time:   2016-09-19 10:53:53 UTC
Revision:     13f38ba699afd86d7cdc4ed8fd7dd3960c0b1f97

Groovy:       2.4.7
Ant:          Apache Ant(TM) version 1.9.6 compiled on June 29 2015
JVM:          1.8.0_91 (Oracle Corporation 25.91-b15)
OS:           Windows 10 10.0 amd64
```

图 1-20

• 搭建 ZooKeeper 环境

ZooKeeper 是一个分布式的分布式应用程序协调服务，很多分布式系统都依赖与 ZooKeeper 集群实现分布式系统间的协调调度，例如：HDFS 2.x、Hbase、Kafka 以及新兴的数据库中间件 MyCat，ZooKeeper 已经成为现代分布式系统的标配。通过 ZooKeeper 可以实现很多高级功能，例如：统一命名服务、统一配置管理、分布式锁、分布式队列等。

Kafka 使用 ZooKeeper 集群管理元数据，例如：记录 Topic 名称、分区以及其副本分配等信息，用户权限控制的相关数据等。Kafka 还会在 ZooKeeper 的某些上添加相应的监听器，用于监听集群的状态，例如：集群中所有 Broker 通过 ZooKeeper 监听 Controller Leader 的状态。我们也可以通过 Kafka 自带的脚本修改 ZooKeeper 触发相应操作，例如：“优先副本”选举。

下面我们开始搭建 ZooKeeper 的环境。从 ZooKeeper 的官网下载其二进制压缩包，之后解压缩，本书使用的版本是 3.4.9，单机模式（毕竟重点是 Kafka，而且读者的个人计算机资源有限，若使用集群模式，请读者自行参考相关资料）。

解压后，将 %ZOOKEEPER%/conf/zoo_sample.cfg 文件复制一份，重命名为 zoo.cfg。修改 zoo.cfg 配置文件，这里主要就是修改 dataDir 配置项，此配置指向 ZooKeeper 存储数据的目录，其他配置可以不进行修改，如图 1-21 所示。

```
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sake.
dataDir=D:\\Program\\zookeeper-3.4.9\\data
```

图 1-21

由于 Kafka 使用依赖于 ZooKeeper 服务，在启动 Kafka 之前，需要先使用 zkServer 命令，启动上文已配置好的 ZooKeeper 服务，如图 1-22 所示。

```
C:\WINDOWS\system32>zkServer
C:\WINDOWS\system32>call "D:\Program\Java\jdk1.8.0.91\bin\java -Dzookeeper.log.dir=D:\Program\zookeeper-3.4.9\bin\.. -Dzookeeper.root.logger=INFO, CONSOLE
-cp "D:\Program\zookeeper-3.4.9\bin\..\build\classes;D:\Program\zookeeper-3.4.9\bin\..\build\lib\*;D:\Program\zookeeper-3.4.9\bin\..\*;D:\Program\zookeeper
-3.4.9\bin\..\lib\*;D:\Program\zookeeper-3.4.9\bin\..\conf" org.apache.zookeeper.server.quorum.QuorumPeerMain "D:\Program\zookeeper-3.4.9\bin\..\conf\zoo.cfg"

2016-11-29 14:30:13,087 [myid:] - INFO [main:ZooKeeperServer#815] - tickTime set to 2000
2016-11-29 14:30:13,087 [myid:] - INFO [main:ZooKeeperServer#824] - minSessionTimeout set to -1
2016-11-29 14:30:13,089 [myid:] - INFO [main:ZooKeeperServer#833] - maxSessionTimeout set to -1
2016-11-29 14:30:13,167 [myid:] - INFO [main:NIOServerCnxnFactory#89] - binding to port 0.0.0.0/0.0.0.0:2181
```

图 1-22

• Kafka 源码构建

本书以 kafka-0.10.0.1 版本为基础进行源码分析。首先，从 Apache Kafka 官网下载

其源码包 kafka-0.10.0.1-src.tgz，解压。使用命令行导航到 Kafka 源代码根目录下，使用 gradle idea 命令进行构建（如果读者希望构建 Eclipse 工程，请使用 gradle eclipse 命令构建），构建过程中会从网上下载各种依赖包，时间会有点长，请耐心等待。最终输出 BUILD SUCCESSFUL 字样，表示构建成功，如图 1-23 所示。

```
D:\kafka-0.10.1.0-src>gradle idea
Starting a Gradle Daemon (subsequent builds will be faster)
Download https://repo1.maven.org/maven2/org/scoverage/gradle-scoverage/2.1.0/gradle-scoverage-2.1.0.pom
Download https://repo1.maven.org/maven2/org/scoverage/gradle-scoverage/2.1.0/gradle-scoverage-2.1.0.jar
...
:streams:examples:idea
BUILD SUCCESSFUL
Total time: 1 mins 21.094 secs
```

图 1-23

• 安装 Scala 插件

本书使用 IntelliJ IDEA 进行源码分析，并安装了其对应的 Scala 插件。本书使用的 Scala 插件版本是 scala-intellij-bin-2016.2.1.zip，请读者选择合适的插件版本。建议使用离线安装方式，安装方法如图 1-24 所示。

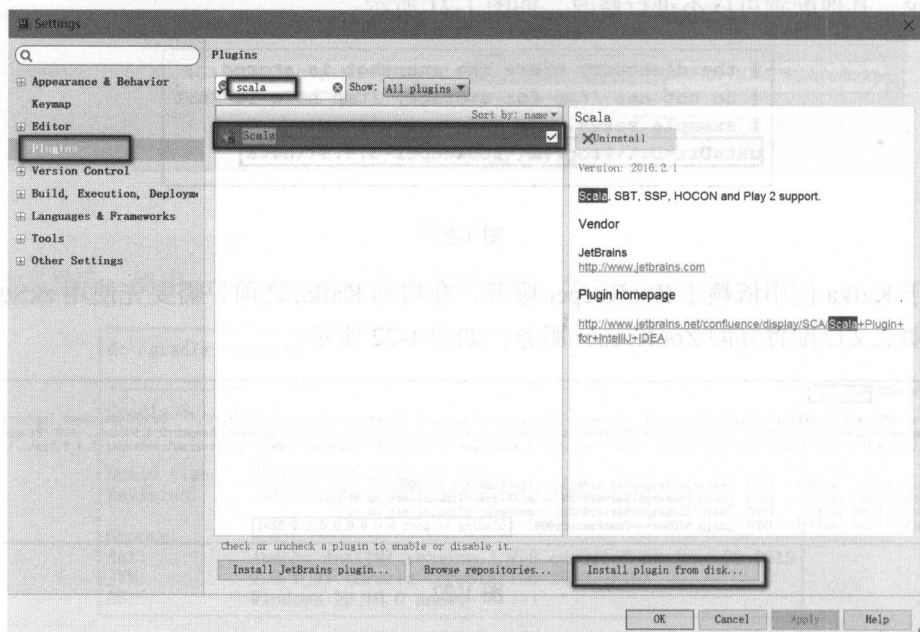


图 1-24

• 配置、启动 Kafka

在 Kafka 服务端使用 log4j 输出日志，启动前需要把 log4j.properties 配置文件放置到 src/main/scala 路径下，然后运行程序，这样才能正确输出日志信息。此 log4j.properties 文件可以从 config 目录中获取，如图 1-25 所示。

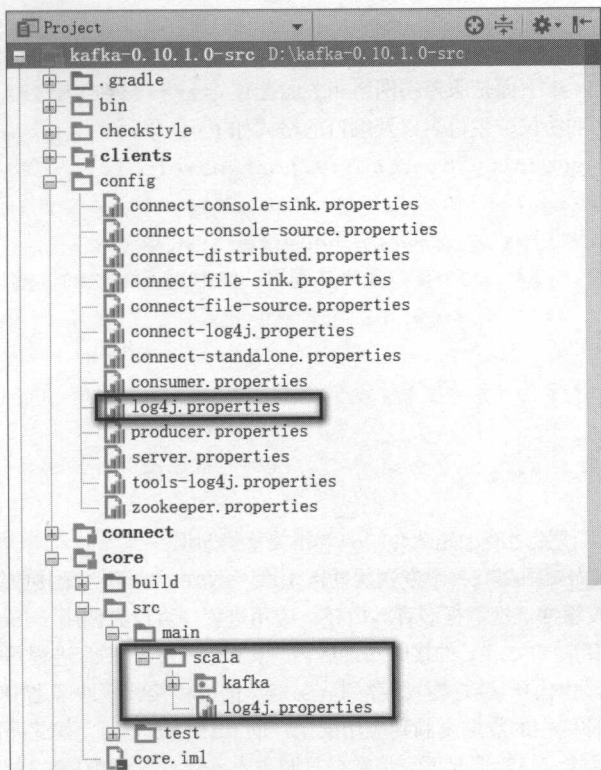


图 1-25

server.properties 是 Kafka 的主要配置文件，下面简单介绍其中的相关配置项的含义。必须修改的配置项就是 log.dirs，其他配置读者可以根据需求自行修改。

```
# 每一个 Broker 在集群中的唯一标识。即使 Broker 的 IP 地址发生了变化，broker.id 只要没变，
# 则不会影响 consumers 的消息情况
broker.id=0
# 是否允许 Topic 被删除。如果是 false，使用管理工具删除 Topic 的时候，Kafka 并不会处
# 理此操作
# delete.topic.enable=true

# Kafka 服务端是否可以根据请求自动创建 Topic，默认是 true。如果打开此选项，下面三
```

```

# 种请求会触发 Topic 自动创建:
# ① Producer 向某个不存在的 Topic 写入消息
# ② Consumer 从某个不存在的 Topic 读取消息
# ③ Consumer 从某个不存在的 Topic 读取消息
# 建议将此选项设置为 false, 并在使用 Topic 之前手动创建
# auto.create.topics.enable=true

##### 下面是服务端网络相关的配置 #####
# Kafka Server 使用的协议、主机名以及端口的格式如下:
# listeners = security_protocol://host_name:port
# 参考示例:
# listeners = PLAINTEXT://your.host.name:9092
#listeners=PLAINTEXT://:9092 # 这是默认配置, 使用 PLAINTEXT, 端口是 9092

# 接收请求的线程数
num.network.threads=3
# 执行请求的线程数
num.io.threads=8

# 在介绍下面两个缓冲区设置之前, 先来介绍一下相关背景知识:
# 每个 TCP socket 在内核中都有一个发送缓冲区 (SO_SNDBUF) 和一个接收缓冲区
# (SO_RCVBUF)。接收缓冲区把数据缓存入内核, 应用进程一直没有调用 read 进行读取的话,
# 此数据会一直缓存在相应 socket 的接收缓冲区内。再啰嗦一点, 不管进程是否读取 socket,
# 对端发来的数据都会经由内核接收并且缓存到 socket 的内核接收缓冲区之中。read 所做的
# 工作, 就是把内核缓冲区中的数据复制到应用层用户的 buffer 里面, 仅此而已。进程调用
# send 发送数据的时候, 一般情况下, 将数据复制进入 socket 的内核发送缓冲区之中, 然
# 后 send 便会在上层返回。换句话说, send 返回之时, 数据不一定会发送到对端去, send
# 仅仅是把应用层 buffer 的数据复制进 socket 的内核发送 buffer 中
# TCP 连接的 SO_SNDBUF 缓冲区大小, 默认 102400, 单位是字节
# 如果是 -1, 就使用操作系统的默认值
socket.send.buffer.bytes=102400
# TCP 连接的 SO_RCVBUF 缓冲区大小, 默认 102400, 单位是字节
# 如果是 -1, 就使用操作系统的默认值
socket.receive.buffer.bytes=102400
# 请求的最大长度
socket.request.max.bytes=104857600

##### 下面是存储 log 相关的配置 #####

```



```

# 用于存储 log 文件的目录，可以将多个目录通过逗号分隔，形成一个目录列表
log.dirs=/tmp/kafka-logs

# 每个 Topic 默认的 partition 数量，默认值是 1
num.partitions=1

# 用来恢复 log 文件以及关闭时将 log 数据刷新到磁盘的线程数量，每个目录对应
# num.recovery.threads.per.data.dir 个线程
num.recovery.threads.per.data.dir=1

##### 下面是 log 文件刷盘的相关配置 #####
# 每隔多少个消息触发一次 flush 操作，将内存中的消息刷新到硬盘上
#log.flush.interval.messages=10000
# 每隔多少毫秒触发一次 flush 操作，将内存中的消息刷新到硬盘上
#log.flush.interval.ms=1000
# 上面这两个配置是全局的，可以在 Topic 中重新设置，并覆盖这两个配置

##### 下面是 log 相关的“保存策略”的配置 #####
# 注意：下面有两种配置，一种是基于时间的策略，另一种是基于日志文件大小的策略，两种
# 策略同是配置的话，只要满足其中一种策略，则触发 Log 删除的操作。删除操作总是先删除
# 最旧的日志

# 消息在 Kafka 中保存的时间，168 小时之前的 log，可以被删除掉
log.retention.hours=168

# 当剩余空间低于 log.retention.bytes 字节，则开始删除 log
#log.retention.bytes=1073741824

# segment 日志文件大小的上限值。当超过这个值时，会创建新的 segment 日志文件
# segment 文件的相关信息在后面介绍
log.segment.bytes=1073741824
# 每隔 300000ms，logcleaner 线程将检查一次，看是否符合上述保留策略的消息可以被删除
log.retention.check.interval.ms=300000

##### ZooKeeper 的相关配置 #####
# Kafka 依赖的 ZooKeeper 集群地址，可以配置多个 ZooKeeper 地址，使用逗号隔开
zookeeper.connect=localhost:2181

```

```
# ZooKeeper 连接的超时时间
zookeeper.connection.timeout.ms=6000
```

- 配合 Kafka 的启动参数

Kafka 服务端的入口类是 `kafka.Kafka`。除了指定入口类，还需要指定 `server.properties` 配置文件所在的位置，即在 Kafka 源代码的 `config` 目录下，如图 1-26 所示。

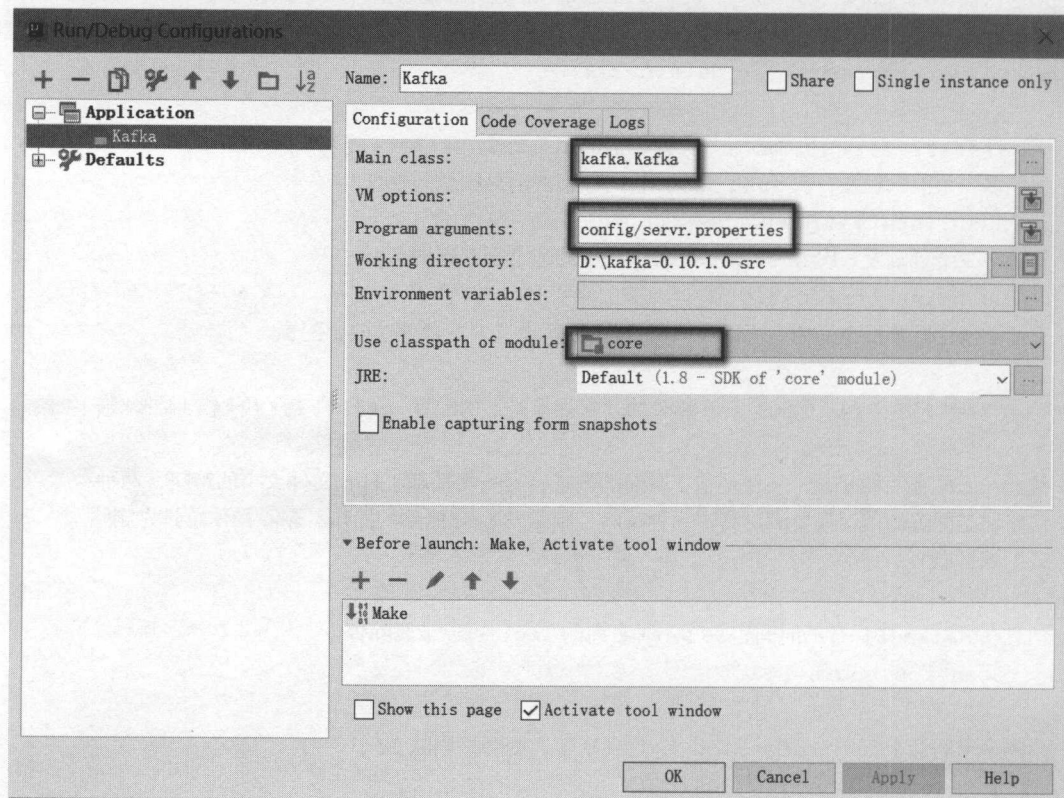


图 1-26

第一次启动时，IntelliJ IDEA 会重新编译整个项目，编译完成后启动。得到的日志如图 1-27 所示。

```

D:\Program\Java\jdk1.8.0.91\bin\java ...
[2016-11-29 15:09:30,615] INFO KafkaConfig values:
    advertised.host.name = null
    advertised.listeners = null
    advertised.port = null
    authorizer.class.name =
    auto.create.topics.enable = true
    auto.leader.rebalance.enable = true

...

[2016-11-29 15:09:32,201] INFO Registered broker 0 at path /brokers/ids/0 with addresses: PLAINTEXT -> EndPoint(DESKTOP-GS4NOSS,9092,PLAINTEXT) (kafka.utils.ZkUtils)
[2016-11-29 15:09:32,202] WARN No meta.properties file under dir D:\tmp\kafka-logs\meta.properties (kafka.server.BrokerMetadataCheckpoint)
[2016-11-29 15:09:32,221] INFO New leader is 0 (kafka.server.ZookeeperLeaderElector$LeaderChangeListener)
[2016-11-29 15:09:32,246] WARN Error while loading kafka-version.properties :null (org.apache.kafka.common.utils.AppInfoParser)
[2016-11-29 15:09:32,247] INFO Kafka version : unknown (org.apache.kafka.common.utils.AppInfoParser)
[2016-11-29 15:09:32,247] INFO Kafka commitId : unknown (org.apache.kafka.common.utils.AppInfoParser)
[2016-11-29 15:09:32,248] INFO [Kafka Server 0], started (kafka.server.KafkaServer)

```

图 1-27

• 验证

为了验证上文配置的源码环境是否成功，可以使用 Kafka 二进制包中自带的三个脚本进行验证，分别是：kafka-topics 用于创建 Topic；kafka-console-producer 是一个命令行 Producer；kafka-console-consumer 是一个命令行 Consumer。

首先，使用 kafka-topics.bat 创建一个示例 Topic——test，其中 partition 和 replication-factor 都为 1，输出如图 1-28 所示。

```

C:\WINDOWS\system32>kafka-topics.bat --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic test
Created topic "test".

```

图 1-28

之后，启动两个命令行窗口，分别执行下面两行命名，一个作为生产者，另一个作为消费者：

```

kafka-console-producer.bat --broker-list localhost:9092 --topic test
kafka-console-consumer.bat --zookeeper localhost:2181 --topic test

```

启动完成后，得到如图 1-29 所示的两个窗口。

```
CA 管理员: Command Prompt - kafka-console-consumer.bat --zookeeper localhost:2181 --topic test
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>kafka-console-consumer.bat --zookeeper localhost:2181 --topic test

CA 管理员: Command Prompt - kafka-console-producer.bat --broker-list localhost:9092 --topic test
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>kafka-console-producer.bat --broker-list localhost:9092 --topic test
```

图 1-29

在生产者的命令行窗口中输入任意内容，回车后，在消费者命令行窗口中能看到相应消息。输出如图 1-30 所示，这就表示我们搭建的 Kafka 源码环境可以正常运行了。安装完成之后，读者可以直接在 IDE 中进行 Debug，分析 Kafka 的工作流程了。

```
CA 管理员: Command Prompt - kafka-console-consumer.bat --zookeeper localhost:2181 --topic test
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>kafka-console-consumer.bat --zookeeper localhost:2181 --topic test
abcdefghijklmn

CA 管理员: Command Prompt - kafka-console-producer.bat --broker-list localhost:9092 --topic test
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\WINDOWS\system32>kafka-console-producer.bat --broker-list localhost:9092 --topic test
abcdefghijklmn
```

图 1-30

本章小结

本章首先对 Kafka 的背景、关键特性以及应用场景做了简单介绍。之后，介绍了笔者在实践中遇到的一个以 Kafka 为中心的案例，并分析了在此案例中选择使用 Kafka 的具体原因和 Kafka 的关键作用。然后介绍了 Kafka 中的核心概念，希望读者对 Kafka 的整个架构设计有宏观的了解。最后，介绍了 Kafka 源码调试环境的搭建，简单介绍了 server.properties 配置文件中的关键配置项。

第2章

生产者

Kafka 在实际应用中，经常被用作高性能、可扩展的消息中间件。Kafka 自定义了一套网络协议，只要遵守这套协议的格式，就可以向 Kafka 发送消息，也可以从 Kafka 中拉取消息。在实践生产过程中，一套 API 封装良好、灵活易用的客户端可以避免开发人员重复劳动，提高开发效率，也可以提高程序的健壮性和可靠性。Kafka 提供了 Java 版本的生产者的实现——KafkaProducer，使用 KafkaProducer 的 API 可以轻松实现同步 / 异步发送消息、批量发送、超时重发等复杂的功能，在业务模块向 Kafka 写入消息时，KafkaProducer 就显得必不可少。

现在，Kafka 的爱好者已经使用多种语言（诸如 C++、Java、Python、Go 等）实现了 Kafka 的客户端。如果读者使用其他语言，可以到 Kafka 官方网站的 wiki（<https://cwiki.apache.org/confluence/display/KAFKA/Clients>）查找相关资料。

在 Kafka core 模块的 kafka.producer 包中，还保留着旧版本的生产者客户端的实现（Scala 实现），此客户端已经被标记为“废弃”，不再推荐大家使用了。新版本的生产者客户端实现 KafkaProducer（Java 实现）在 Kafka clients 模块的 org.apache.kafka.clients.producer 包中。本章将以新版本的 KafkaProducer 为基础展开介绍，学习 KafkaProducer 的使用和相关配置，并深入剖析其核心代码及原理。对于旧版本的客户端本书不做详细介绍。

2.1 KafkaProducer 使用示例

首先通过一个示例展示 KafkaProducer 的使用方法。在下面的示例程序中，我们使

用 `KafkaProducer` 实现向 `Kafka` 发送消息的功能。在示例程序中，首先将 `KafkaProducer` 使用到的配置项写入到 `Properties` 中，每项配置的具体含义在注释中进行解释。之后以此 `Properties` 对象为参数构造 `KafkaProducer` 对象。最后通过 `KafkaProducer` 的 `send()` 方法完成消息发送。

```
public class ProducerDemo {
    public static void main(String[] args) {
        boolean isAsync = args.length == 0 ||
            // 消息的发送方式：异步发送还是同步发送
            !args[0].trim().equalsIgnoreCase("sync");

        Properties props = new Properties();
        // Kafka 服务端的主机名和端口号
        props.put("bootstrap.servers", "localhost:9092");
        props.put("client.id", "DemoProducer");// 客户端的 ID
        // 消息的 key 和 value 都是字节数组，为了将 Java 对象转化为字节数组，可以配置
        // “key.serializer” 和 “value.serializer” 两个序列化器，完成转化
        props.put("key.serializer",
            "org.apache.kafka.common.serialization.
IntegerSerializer");
        // StringSerializer 用来将 String 对象序列化成字节数组
        props.put("value.serializer",
            "org.apache.kafka.common.serialization.StringSerializer");
        KafkaProducer producer = new KafkaProducer<>(props); // 生产者的核心类
        String topic = "test"; // 向指定的 test 这个 Topic 发送消息

        int messageNo = 1; // 消息的 key
        while (true) {
            String messageStr = "Message_" + messageNo; // 消息的 value
            long startTime = System.currentTimeMillis();
            if (isAsync) { // 异步发送消息
                // 第一个参数是 ProducerRecord 类型的对象，封装了目标 Topic、消息的
                // key、消息的 value
                // 第二个参数是一个 Callback 对象，当生产者接收到 Kafka 发来的 ACK 确
                // 认消息的时候，会调用此 Callback 对象的 onCompletion() 方法，实现
                // 回调功能
            }
        }
    }
}
```

```

        producer.send(new ProducerRecord<>(topic, messageNo,
messageStr),
                                new DemoCallBack(startTime, messageNo,
messageStr));
    } else { // 同步发送消息
        try {
            // KafkaProducer.send() 方法的返回值类型是 Future<RecordMetadata>
            // 这里通过 Future.get() 方法, 阻塞当前线程, 等待 Kafka 服务端的 ACK 响应
            producer.send(new ProducerRecord<>(topic, messageNo,
messageStr)).get();
            System.out.println("Sent message: (" + messageNo
+ ", " + messageStr + ")");
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
    ++messageNo; // 递增消息的 key
}

}

}

class DemoCallBack implements Callback { // 回调对象
    private final long startTime; // 开始发送消息的时间戳
    private final int key; // 消息的 key
    private final String message; // 消息的 value

    public DemoCallBack(long startTime, int key, String message) {
        this.startTime = startTime;
        this.key = key;
        this.message = message;
    }

    /**
     * 生产者成功发送消息, 收到 Kafka 服务端发来的 ACK 确认消息后, 会调用此回调函数
     * @param metadata 生产者发送的消息的元数据, 如果发送过程中出现异常, 此参数为 null

```

```

    * @param exception 发送过程中出现的异常，如果发送成功，则此参数为 null
    */
    public void onCompletion(RecordMetadata metadata, Exception exception)
    {
        long elapsedTime = System.currentTimeMillis() - startTime;
        if (metadata != null) {
            // RecordMetadata 中包含了分区信息、Offset 信息等
            System.out.println("message(" + key + ", " + message + ") sent
to partition("
                                + metadata.partition() + "), " +
                                "offset(" + metadata.offset() + ") in " + elapsedTime
+ " ms");
        } else {
            exception.printStackTrace();
        }
    }
}

```

只要参考注释并注意 `KafkaProducer` 的相关 API，对于熟悉 Java 的读者来说，上面这段程序并没有太大的难度。随着后面分析，读者可能会发现这简单的 API 之后的巧妙设计。

2.2 KafkaProducer 分析

了解了 `KafkaProducer` 的基本使用方法后，本节开始深入分析 `KafkaProducer` 的原理和实现。在图 2-1 中简略描述了 `KafkaProducer` 发送消息的整个流程，希望读者对 `KafkaProducer` 有一个大致的了解。

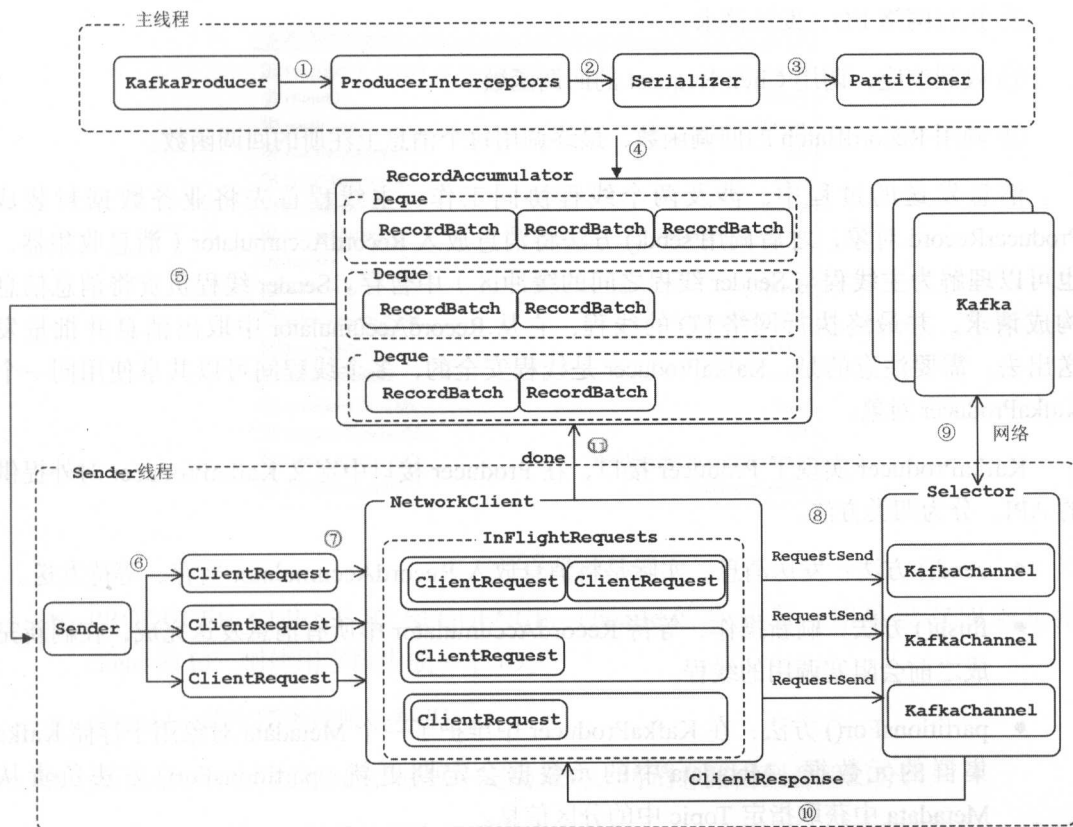


图 2-1

下面简述图 2-1 中每个步骤的操作，让读者对整个过程有宏观的了解：

- ① **ProducerInterceptors** 对消息进行拦截。
- ② **Serializer** 对消息的 key 和 value 进行序列化。
- ③ **Partitioner** 为消息选择合适的 Partition。
- ④ **RecordAccumulator** 收集消息，实现批量发送。
- ⑤ **Sender** 从 **RecordAccumulator** 获取消息。
- ⑥ 构造 **ClientRequest**。
- ⑦ 将 **ClientRequest** 交给 **NetworkClient**，准备发送。
- ⑧ **NetworkClient** 将请求放入 **KafkaChannel** 的缓存。

⑨ 执行网络 I/O，发送请求。

⑩ 收到响应，调用 `ClientRequest` 的回调函数。

⑪ 调用 `RecordBatch` 的回调函数，最终调用每个消息上注册的回调函数。

消息发送的过程中，涉及两个线程协同工作。主线程首先将业务数据封装成 `ProducerRecord` 对象，之后调用 `send()` 方法将消息放入 `RecordAccumulator`（消息收集器，也可以理解为主线程与 `Sender` 线程之间的缓冲区）中暂存。`Sender` 线程负责将消息信息构成请求，并最终执行网络 I/O 的线程，它从 `RecordAccumulator` 中取出消息并批量发送出去。需要注意的是，`KafkaProducer` 是线程安全的，多个线程间可以共享使用同一个 `KafkaProducer` 对象。

`KafkaProducer` 实现了 `Producer` 接口，在 `Producer` 接口中定义 `KafkaProducer` 对外提供的 API，分为四类方法。

- `send()` 方法：发送消息，实际是将消息放入 `RecordAccumulator` 暂存，等待发送。
- `flush()` 方法：刷新操作，等待 `RecordAccumulator` 中所有消息发送完成，在刷新完成之前会阻塞调用的线程。
- `partitionsFor()` 方法：在 `KafkaProducer` 中维护了一个 `Metadata` 对象用于存储 Kafka 集群的元数据，`Metadata` 中的元数据会定期更新。`partitionsFor()` 方法负责从 `Metadata` 中获取指定 Topic 中的分区信息。
- `close()` 方法：关闭此 `Producer` 对象，主要操作是设置 `close` 标志，等待 `RecordAccumulator` 中的消息清空，关闭 `Sender` 线程。

还有一个 `metrics()` 方法，用于记录统计信息，与消息发送的流程无关，我们不做详细分析。

了解了 `Producer` 接口的功能之后，我们下面就来分析 `KafkaProducer` 的具体实现。首先，介绍 `KafkaProducer` 中比较重要的字段，在后面分析过程中，会逐个进行分析，如图 2-2 所示。

KafkaProducer	
PRODUCER_CLIENT_ID_SEQUENCE	AtomicInteger
clientId	String
partitioner	Partitioner
maxRequestSize	int
totalMemorySize	long
metadata	Metadata
accumulator	RecordAccumulator
sender	Sender
ioThread	Thread
compressionType	CompressionType
keySerializer	Serializer<K>
valueSerializer	Serializer<V>
maxBlockTimeMs	long
requestTimeoutMs	int
interceptors	ProducerInterceptors<K, V>
producerConfig	ProducerConfig

图 2-2

- **PRODUCER_CLIENT_ID_SEQUENCE**: `clientId` 的生成器，如果没有明确指定 `client` 的 `Id`，则使用字段生成一个 `ID`。
- **clientId**: 此生产者的唯一标识。
- **partitioner**: 分区选择器，根据一定的策略，将消息路由到合适的分区。
- **maxRequestSize**: 消息的最大长度，这个长度包含了消息头、序列化后的 `key` 和序列化后的 `value` 的长度。
- **totalMemorySize**: 发送单个消息的缓冲区大小。
- **accumulator**: `RecordAccumulator`，用于收集并缓存消息，等待 `Sender` 线程发送。
- **sender**: 发送消息的 `Sender` 任务，实现了 `Runnable` 接口，在 `ioThread` 线程中执行。
- **ioThread**: 执行 `Sender` 任务发送消息的线程，称为“`Sender` 线程”。
- **compressionType**: 压缩算法，可选项有 `none`、`gzip`、`snappy`、`lz4`。这是针对 `RecordAccumulator` 中多条消息进行的压缩，所以消息越多，压缩效果越好。
- **keySerializer**: `key` 的序列化器。
- **valueSerializer**: `value` 的序列化器。
- **Metadata metadata**: 整个 `Kafka` 集群的元数据。

- `maxBlockTimeMs`: 等待更新 Kafka 集群元数据的最大时长。
- `requestTimeoutMs`: 消息的超时时间, 也就是从消息发送到收到 ACK 响应的最长时长。
- `interceptors`: `ProducerInterceptor` 集合, `ProducerInterceptor` 可以在消息发送之前对其进行拦截或修改; 也可以先于用户的 `Callback`, 对 ACK 响应进行预处理。
- `producerConfig`: 配置对象, 使用反射初始化 `KafkaProducer` 配置的相对对象。

在 `KafkaProducer` 的构造函数中, 会初始化上面介绍的字段, 其中有几个需要注意:

```
private KafkaProducer(ProducerConfig config, Serializer<K> keySerializer,
                      Serializer<V> valueSerializer) {
    ... ..
    // 通过反射机制实例化配置的 partitioner 类、keySerializer 类、valueSerializer 类
    this.partitioner = config.getConfiguredInstance();
    this.keySerializer = config.getConfiguredInstance(...);
    this.valueSerializer = config.getConfiguredInstance(...);

    // 创建并更新 Kafka 集群的元数据
    this.metadata = new Metadata(...);
    List<InetSocketAddress> addresses = ClientUtils.parseAndValidateAddresses(
        config.getList(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG));
    this.metadata.update(Cluster.bootstrap(addresses), time.
        milliseconds());

    // 创建 RecordAccumulator
    this.accumulator = new RecordAccumulator(
        // batch.size 参数指定的每个 RecordBatch 的大小, 单位是字节
        config.getInt(ProducerConfig.BATCH_SIZE_CONFIG),
        this.totalMemorySize, this.compressionType,
        config.getLong(ProducerConfig.LINGER_MS_CONFIG),
        retryBackoffMs, metrics, time
    );

    // 创建 NetworkClient, 这个是 KafkaProducer 网络 I/O 的核心, 在后面会详细介绍
    NetworkClient client = new NetworkClient(new Selector(...), ...);
    this.sender = new Sender(...);
```

```
// 启动 Sender 对应的线程
this.ioThread = new KafkaThread(ioThreadName, this.sender, true);
this.ioThread.start()
... ..
}
```

KafkaProducer 构造完成之后，我们来关注 KafkaProducer 的 send() 方法。图 2-3 展示了整个 send() 方法的调用流程。

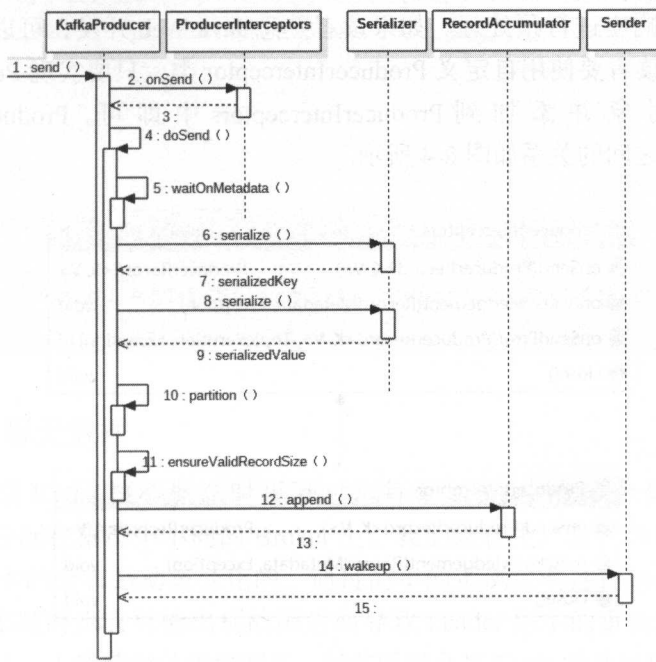


图 2-3

图 2-3 中的关键步骤如下：

- 调用 `ProducerInterceptors.onSend()` 方法，通过 `ProducerInterceptor` 对消息进行拦截或修改。
- 调用 `waitOnMetadata()` 方法获取 Kafka 集群的信息，底层会唤醒 Send 线程更新 Metadata 中保存的 Kafka 集群元数据。
- 调用 `Serializer.serialize()` 方法序列化消息的 key 和 value。
- 调用 `partition()` 为消息选择合适的分区。

- 调用 `RecordAccumulator.append()` 方法，将消息追加到 `RecordAccumulator` 中。
- 唤醒 `Sender` 线程，由 `Sender` 线程将 `RecordAccumulator` 中缓存的消息发送出去。

2.2.1 ProducerInterceptors&ProducerInterceptor

`ProducerInterceptors` 是一个 `ProducerInterceptor` 集合，其 `onSend` 方法、`onAcknowledgement` 方法、`onSendError` 方法，实际上是循环调用其封装的 `ProducerInterceptor` 集合的对应方法。

`ProducerInterceptor` 对象可以在消息发送之前对其进行拦截或修改，也可以先于用户的 `Callback`，对 `ACK` 响应进行预处理。如果读者熟悉 Java Web 开发，可以将其与 `Filter` 的功能做类比。如果读者要使用自定义 `ProducerInterceptor` 类，只要实现 `ProducerInterceptor` 接口，创建其对象并添加到 `ProducerInterceptors` 中即可。`ProducerInterceptors` 与 `ProducerInterceptor` 之间的关系如图 2-4 所示。

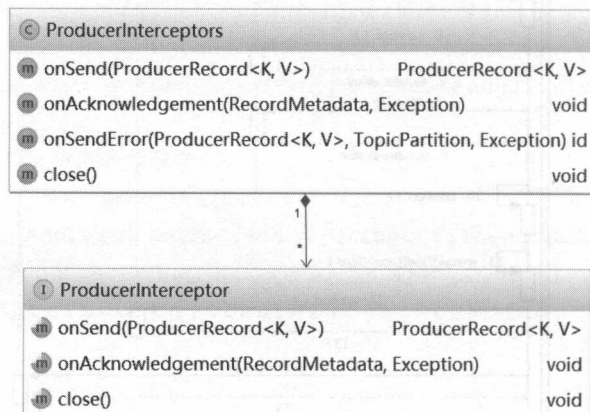


图 2-4

下面提供一个 `ProducerInterceptor` 的示例实现，供读者参考：

```

public class ProducerInterceptorDemo implements ProducerInterceptor<Integer,
String> {

    @Override
    public ProducerRecord<Integer, String> onSend(
        ProducerRecord<Integer, String> record) {
        if (record.key() % 2 == 0) return record; // 过滤掉 key 为奇数的消息
    }
}
  
```

```

        return null;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception
exception) {
        if (metadata != null && "test".equals(metadata.topic())
            && metadata.partition() == 0) {
            // 对于正常返回, Topic 为 test 且分区编号为 0 的消息的返回信息进行输出
            System.out.println(metadata.toString());
        }
    }

    @Override
    public void close() {}

    @Override
    public void configure(Map<String, ?> configs) { // 用来初始化此类的方法 }
}

```

2.2.2 Kafka 集群元数据

在第1章介绍 Kafka 核心概念时提到过, 每个 Topic 中有多个分区, 这些分区的 Leader 副本可以分配在集群中不同的 Broker 上。我们站在生产者的角度来看, 分区的数量以及 Leader 副本的分布是动态变化的。通过简单的示例说明这种动态变化: 在运行过程中, Leader 副本随时都有可能出现故障进而导致 Leader 副本的重新选举, 新的 Leader 副本会在其他 Broker 上继续对外提供服务。当需要提高某 Topic 的并行处理消息的能力时, 我们可以通过增加其分区的数量来实现。当然, 还有别的方式导致这种动态变化, 例如, 手动触发“优先副本”选举等。

请读者先回顾在本章开始给出的 KafkaProducer 示例, 我们创建的 ProducerRecord 中只指定了 Topic 的名称, 并未明确指定分区编号。KafkaProducer 要将此消息追加到指定 Topic 的某个分区的 Leader 副本中, 首先需要知道 Topic 的分区数量, 经过路由后确定目标分区, 之后 KafkaProducer 需要知道目标分区的 Leader 副本所在服务器的地址、端口等信息, 才能建立连接, 将消息发送到 Kafka 中。因此, 在 KafkaProducer 中维护了 Kafka 集群的元数据, 这些元数据记录了: 某个 Topic 中有哪几个分区, 每个分区的 Leader 副本分配哪个节点上, Follower 副本分配哪些节点上, 哪些副本在 ISR 集合中以及这些节点的

网络地址、端口。

在 `KafkaProducer` 中, 使用 `Node`、`TopicPartition`、`PartitionInfo` 这三个类封装了 Kafka 集群的相关元数据, 其主要字段如图 2-5 所示。

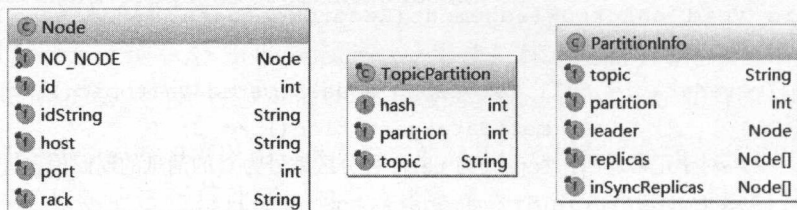


图 2-5

- `Node` 表示集群中的一个节点, `Node` 记录这个节点的 `host`、`ip`、`port` 等信息。
- `TopicPartition` 表示某 `Topic` 的一个分区, 其中的 `topic` 字段是 `Topic` 的名称, `partition` 字段则此分区在 `Topic` 中的分区编号 (ID)。
- `PartitionInfo` 表示一个分区的详细信息。其中 `topic` 字段和 `partition` 字段的含义与 `TopicPartition` 中的相同, 除此之外, `leader` 字段记录了 Leader 副本所在节点的 `id`, `replica` 字段记录了全部副本所在的节点信息, `inSyncReplicas` 字段记录了 ISR 集合中所有副本所在的节点信息。

通过这三个类的组合, 我们可以完整表示出 `KafkaProducer` 需要的集群元数据。这些元数据保存在了 `Cluster` 这个类中, 并按照不同的映射方式进行存放, 方便查询。`Cluster` 类的核心字段如图 2-6 所示。

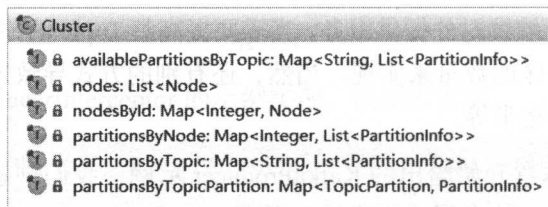


图 2-6

- `nodes`: Kafka 集群中节点信息列表。
- `nodesById`: `BrokerId` 与 `Node` 节点之间对应关系, 方便按照 `BrokerId` 进行索引。
- `partitionsByTopicPartition`: 记录了 `TopicPartition` 与 `PartitionInfo` 的映射关系。

- `partitionsByTopic`: 记录了 Topic 名称和 `PartitionInfo` 的映射关系, 可以按照 Topic 名称查询其中全部分区的详细信息。
- `availablePartitionsByTopic`: Topic 与 `PartitionInfo` 的映射关系, 这里的 `List<PartitionInfo>` 中存放的分区必须是有 Leader 副本的 `Partition`, 而 `partitionsByTopic` 中记录的分区则不一定有 Leader 副本, 因为某些中间状态, 例如 Leader 副本宕机而触发的选举过程中, 分区不一定有 Leader 副本 (在第 4 章中会详细介绍)。
- `partitionsByNode`: 记录了 Node 与 `PartitionInfo` 的映射关系, 可以按照节点 Id 查询其上分布的全部分区的详细信息。

`Cluster` 的方法比较简单, 主要是对上述集合的操作, 方便集群元数据的查询。例如, `partitionsForTopic` 方法:

```
public List<PartitionInfo> partitionsForTopic(String topic) {
    return this.partitionsByTopic.get(topic); // 获取指定 topic 的分区集合
}
```

其余方法这里不再赘述了。值得注意的是, `Node`、`TopicPartition`、`PartitionInfo`、`Cluster` 的所有字段都是 `private final` 修饰的, 且只提供了查询方法, 并未提供任何修改方法, 这就保证了这四个类的对象都是不可变性对象, 它们也就成为了线程安全的对象。

`Metadata` 中封装了 `Cluster` 对象, 并保存 `Cluster` 数据的最后更新时间、版本号 (`version`)、是否需要更新等待信息, 如图 2-7 所示。

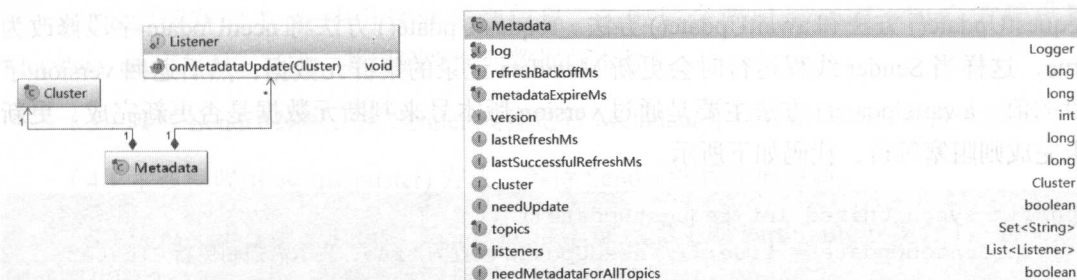


图 2-7

下面介绍一下 `Metadata` 的核心字段。

- `topics`: 记录了当前已知的所有 topic, 在 `cluster` 字段中记录了 Topic 最新的元数据。
- `version`: 表示 Kafka 集群元数据的版本号。Kafka 集群元数据每更新成功一次, `version` 字段的值增 1。通过新旧版本号的比较, 判断集群元数据是否更新完成。

- `metadataExpireMs`: 每隔多久, 更新一次。默认是 300×1000 , 也就是 5 分钟。
- `refreshBackoffMs`: 两次发出更新 Cluster 保存的元数据信息的最小时间差, 默认为 100ms。这是为了防止更新操作过于频繁而造成网络阻塞和增加服务端压力。在 Kafka 中与重试操作有关的操作中, 都有这种“退避 (backoff) 时间”设计的身影。
- `lastRefreshMs`: 记录上一次更新元数据的时间戳 (也包含更新失败的情况)。
- `lastSuccessfulRefreshMs`: 上一次成功更新的时间戳。如果每次都成功, 则 `lastSuccessfulRefreshMs`、`lastRefreshMs` 相等。否则, `lastRefreshMs` > `lastSuccessfulRefreshMs`。
- `cluster`: 记录 Kafka 集群的元数据。
- `needUpdate`: 标识是否强制更新 Cluster, 这是触发 Sender 线程更新集群元数据的条件之一。
- `listeners`: 监听 Metadata 更新的监听器集合。自定义 Metadata 监听实现 `Metadata.Listener.onMetadataUpdate()` 方法即可, 在更新 Metadata 中的 `cluster` 字段之前, 会通知 listener 集合中全部 Listener 对象。
- `needMetadataForAllTopics`: 是否需要更新全部 Topic 的元数据, 一般情况下, `KafkaProducer` 只维护它用到的 Topic 的元数据, 是集群中全部 Topic 的子集。

Metadata 的方法比较简单, 主要是操纵上面的几个字段, 这里着重介绍主线程用到的 `requestUpdate()` 方法和 `awaitUpdate()` 方法。`requestUpdate()` 方法将 `needUpdate` 字段修改为 `true`, 这样当 Sender 线程运行时会更新 Metadata 记录的集群元数据, 然后返回 `version` 字段的值。`awaitUpdate()` 方法主要是通过 `version` 版本号来判断元数据是否更新完成, 更新未完成则阻塞等待, 代码如下所示。

```
public synchronized int requestUpdate() {
    this.needUpdate = true; // needUpdate 设置为 true, 表示需强制更新 Cluster
    return this.version;    // 返回当前 Kafka 集群元数据的版本号
}

public synchronized void awaitUpdate(final int lastVersion, final long
maxWaitMs) {
    ... ..
}
```



```

long begin = System.currentTimeMillis();
long remainingWaitMs = maxWaitMs;
// 比较版本号, 通过版本号比较集群元数据是否更新完成
while (this.version <= lastVersion) {
    // 从下面这行代码可以看出, 主线程与 Sender 通过 wait/notify 同步, 更新元数据的操
    // 作则交给 Sender 线程去完成
    if (remainingWaitMs != 0) wait(remainingWaitMs);
    long elapsed = System.currentTimeMillis() - begin;
    if (elapsed >= maxWaitMs) throw new TimeoutException(...);
    remainingWaitMs = maxWaitMs - elapsed;
}
}

```

读者要注意一点, Metadata 中的字段可以由主线程读、Sender 线程更新, 因此它必须是线程安全的, 这也是上面为什么所有方法都使用 synchronized 同步的原因。Sender 线程的内容在本章后面的小节会详细介绍。

下面回到 KafkaProducer.waitOnMetadata() 方法的分析, 它负责触发 Kafka 集群元数据的更新, 并阻塞主线程等待更新完毕。它的主要步骤是:

(1) 检测 Metadata 中是否包含指定 Topic 的元数据, 若不包含, 则将 Topic 添加到 topics 集合中, 下次更新时会从服务端获取指定 Topic 的元数据。

(2) 尝试获取 Topic 中分区的详细信息, 失败后会调用 requestUpdate() 方法设置 Metadata. needUpdate 字段, 并得到当前元数据版本号。

(3) 唤醒 Sender 线程, 由 Sender 线程更新 Metadata 中保存的 Kafka 集群元数据。

(4) 主线程调用 awaitUpdate() 方法, 等待 Sender 线程完成更新。

(5) 从 Metadata 中获取指定 Topic 分区的详细信息 (即 PartitionInfo 集合)。若失败, 则回到步骤 2 继续尝试, 若等待时间超时, 则抛出异常。

waitOnMetadata() 方法的具体实现如下:

```

private long waitOnMetadata(String topic, long maxWaitMs) throws
InterruptedException {
    // 检查 topics 集合中是否包含指定 Topic
    if (!this.metadata.containsTopic(topic))
        this.metadata.add(topic);

    // 成功获取分区的详细信息
    if (metadata.fetch().partitionsForTopic(topic) != null)
        return 0;

    long begin = time.milliseconds();
    long remainingWaitMs = maxWaitMs;
    while (metadata.fetch().partitionsForTopic(topic) == null) {
        // 设置 needupdate, 获取当前元数据版本号
        int version = metadata.requestUpdate();
        sender.wakeup(); // 唤醒 Sender 线程
        // 阻塞等待元数据更新完毕
        metadata.awaitUpdate(version, remainingWaitMs);

        long elapsed = time.milliseconds() - begin;
        if (elapsed >= maxWaitMs) // 检测超时时间
            throw new TimeoutException("...");
        // 检测权限
        if (metadata.fetch().unauthorizedTopics().contains(topic))
            throw new TopicAuthorizationException(topic);
        remainingWaitMs = maxWaitMs - elapsed;
    }
    return time.milliseconds() - begin;
}

```

2.2.3 Serializer&Deserializer

客户端发送的消息的 key 和 value 都是 byte 数组, Serializer 和 Deserializer 接口提供了将 Java 对象序列化 (反序列化) 为 byte 数组的功能。在 KafkaProducer 中, 根据配置文件, 使用合适的 Serializer。图 2-8 展示了 Serializer 和 Deserializer 接口以及它们的实现类。

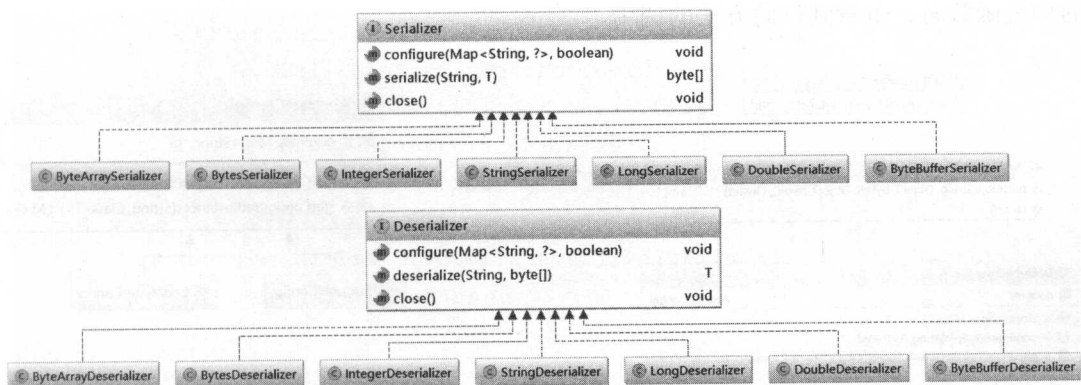


图 2-8

Kafka 已经为我们提供了 Java 基本类型的 Serializer 实现和 Deserializer 实现，我们也可以对 Java 复杂类型的自定义 Serializer 和 Deserializer 实现，只要实现 Serializer 或 Deserializer 接口即可。下面简单介绍 Serializer，Deserializer 是其逆操作。

在 Serializer 接口中，configure() 方法是在执行序列化操作之前的配置，例如，在 StringSerializer.configure() 方法中会选择合适的编码类型（encoding），默认是 UTF-8；IntegerSerializer.configure() 方法则是空实现。serialize() 方法是真正进行序列化的地方，将传入的 Java 对象序列化为 byte[]。close() 方法是在其后的关闭方法，多为空实现。代码比较简单，就不贴出来了，读者可以参考 StringSerializer 和 IntegerSerializer 这两个实现，完成自定义 Serializer。

2.2.4 Partitioner

KafkaProducer.send() 方法的下一步操作是选择消息的分区。在有的应用场景中，由业务逻辑控制每个消息追加到合适的分区中，而有时候业务逻辑并不关心分区的选择。在 KafkaProducer.partition() 方法中，优先根据 ProducerRecord 中 partition 字段指定的序号选择分区，如果 ProducerRecord.partition 字段没有明确指定分区编号，则通过 Partitioner.partition() 方法选择 Partition。

Kafka 提供了 Partitioner 接口的一个默认实现——DefaultPartitioner，继承结构如图 2-9（左）所示，可以看到上面介绍的 ProducerInterceptor 接口也继承了 Configurable 接口。

在创建 KafkaProducer 时传入的 key/value 配置项会保存到 AbstractConfig 的 originals 字段中，如图 2-9（右）所示。AbstractConfig 的核心方法是 getConfiguredInstance() 方法，其主要功能是通过反射机制实例化 originals 字段中指定的类。在前面分析 KafkaProducer

的构造函数时，也看到过此方法的调用。

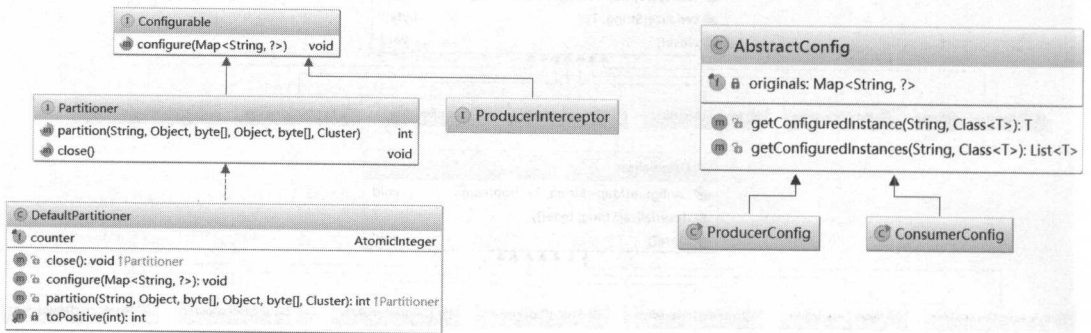


图 2-9

设计 `Configurable` 接口的目的是统一反射后的初始化过程，对外提供统一的初始化接口。在 `AbstractConfig.getConfiguredInstance` 方法中通过反射构造出来的对象，都是通过无参构造函数构造的，需要初始化的字段个数和类型各式各样，`Configurable` 接口的 `configure()` 方法封装了对象初始化过程且只有一个参数（`originals` 字段），这样对外的接口就变得统一。这种设计的小技巧，请读者注意积累。`DefaultPartitioner.configure()` 方法（空实现）继承自 `Configurable` 接口。`close` 方法是在 `Partitioner` 关闭时调用的，也是空实现。

`DefaultPartitioner.partition()` 方法负责在 `ProduceRecord` 中没有明确指定分区编号的时候，为其选择合适的分区：如果消息没有 `key`，会根据 `counter` 与 `Partition` 个数取模来确定分区编号，`count` 不断递增，确保消息不会都发到同一个 `Partition` 里；如果消息有 `key` 的话，则对 `key` 进行 `hash`（使用的是 `murmur2` 这种高效率低碰撞的 `Hash` 算法），然后与分区数量取模，来确定 `key` 所在的分区达到负载均衡。

```

// counter 初始化为一个随机数。注意，这里是 AtomicInteger 对象
private final AtomicInteger counter = new AtomicInteger(new Random().
nextInt());
public int partition(String topic, Object key, byte[] keyBytes,
                    Object value, byte[] valueBytes, Cluster cluster) {
    // 从 Cluster 中获取对应 Topic 的分区信息
    List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
    int numPartitions = partitions.size(); // 分区数量
    if (keyBytes == null) { // 消息没有 key 的情况
        int nextValue = counter.getAndIncrement(); // 递增 counter
        // 选择 availablePartitions
    }
}

```



```

List<PartitionInfo> availablePartitions = cluster
    .availablePartitionsForTopic(topic);
int part = DefaultPartitioner.toPositive(nextValue)
    % availablePartitions.size();
return availablePartitions.get(part).partition();
} else { // 消息有 key 的情况
    return DefaultPartitioner.toPositive(Utils.murmur2(keyBytes))
        % numPartitions;
}
}

```

有的读者可能会觉得奇怪,为什么 counter 不使用 int 类型,而使用 AtomicInteger 类型?这是因为 KafkaProducer 是一个线程安全的类,可能有多个业务线程调用它来发送数据,所以 Partitioner 也必须是线程安全的。从这个角度回顾一下 KafkaProducer 字段,会发现 KafkaProducer 依赖的这些类全部都是线程安全的。

2.3 RecordAccumulator 分析

前面介绍过, KafkaProducer 可以有同步和异步两种方式发送消息,其实两者的底层实现相同,都是通过异步方式实现的。主线程调用 KafkaProducer.send() 方法发送消息的时候,先将消息放到 RecordAccumulator 中暂存,然后主线程就可以从 send() 方法中返回了,此时消息并没有真正地发送给 Kafka,而是缓存在了 RecordAccumulator 中。之后,业务线程通过 KafkaProducer.send() 方法不断向 RecordAccumulator 追加消息,当达到一定的条件,会唤醒 Sender 线程发送 RecordAccumulator 中的消息。

下面我们就来介绍 RecordAccumulator 的结构。首先需要注意的是, RecordAccumulator 至少有一个业务线程和一个 Sender 线程并发操作,所以必须是线程安全的。

RecordAccumulator 中有一个以 TopicPartition 为 key 的 ConcurrentMap, 每个 value 是 ArrayDeque<RecordBatch> (ArrayDeque 并不是线程安全的集合,后面会详细介绍其加锁处理过程), 其中缓存了发往对应 TopicPartition 的消息。每个 RecordBatch 拥有一个 MemoryRecords 对象的引用。MemoryRecords 才是消息最终存放的地方。这三个类的依赖关系如图 2-10 所示。

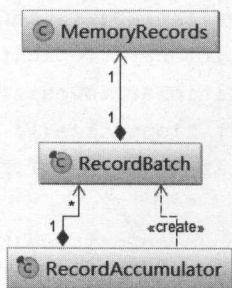


图 2-10

2.3.1 MemoryRecords

大体了解了 RecordAccumulator 的结构之后，我们就从最底层的 MemoryRecords 开始分析。MemoryRecords 表示的是多个消息的集合，其中封装了 Java NIO ByteBuffer 用来保存消息数据，Compressor 用于对 ByteBuffer 中的消息进行压缩，以及其他控制字段。如图 2-11（左）所示，有四个字段比较重要，简单介绍一下。

- **buffer**：用于保存消息数据的 Java NIO ByteBuffer。
- **writeLimit**：记录 buffer 字段最多可以写入多少个字节的数据。
- **compressor**：压缩器，对消息数据进行压缩，将压缩后的数据输出到 buffer。
- **writable**：此 MemoryRecords 对象是只读的模式，还是可写模式。在 MemoryRecords 发送前时，会将其设置成只读模式。

在 Compressor 比较重要的字段和方法如图 2-11（右）所示，有两个输出流类型的字段，分别是 **bufferStream** 和 **appendStream**。前者是在 buffer 上建立的 **ByteBufferOutputStream**（Kafka 自己提供的实现）对象，**ByteBufferOutputStream** 继承了 **java.io.OutputStream**，封装了 **ByteBuffer**，当写入数据超出 **ByteBuffer** 容量时，**ByteBufferOutputStream** 会进行自动扩容；后者是 **DataOutputStream** 类型，它对前者进行了一层装饰，为其添加了压缩的功能。MemoryRecords 中的 Compressor 的压缩类型是由“**compression.type**”配置参数指定的，即 **KafkaProducer.compressionType** 字段的值。下面来分析一下创建压缩流的方式，目前 **KafkaProducer** 支持 GZIP、SNAPPY、LZ4 三种压缩方式。

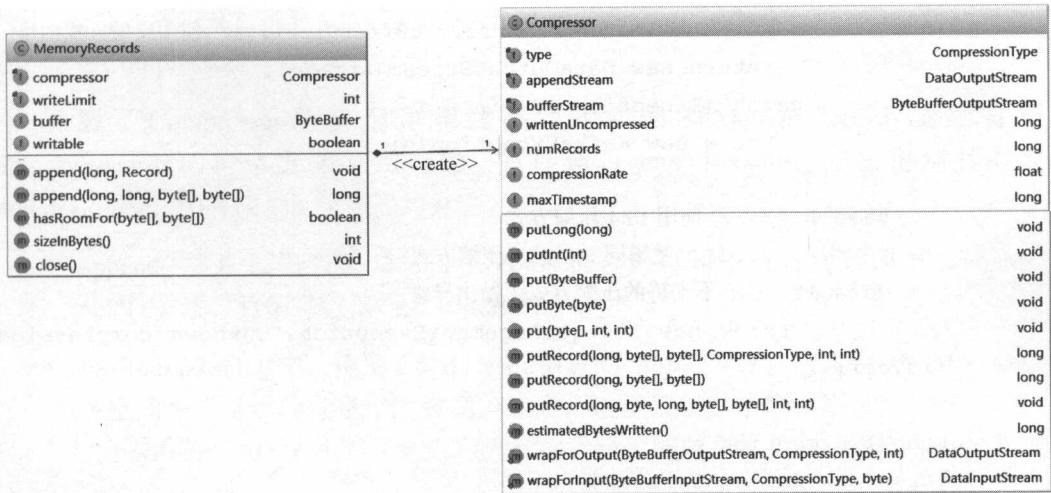


图 2-11

```

public Compressor(ByteBuffer buffer, CompressionType type) {
    this.type = type; // 从KafkaProducer传递过来的压缩类型
    ... ..
    bufferSize = new ByteBufferOutputStream(buffer);
    // 下面根据压缩类型创建合适的压缩流
    appendStream = wrapForOutput(bufferStream, type, COMPRESSION_DEFAULT_
BUFFER_SIZE);
}

public static DataOutputStream wrapForOutput(ByteBufferOutputStream
buffer,
    CompressionType type, int bufferSize) {
    try {
        switch (type) { // 根据不同的类型选择创建不同压缩流
            case NONE: // 不压缩的方式
                return new DataOutputStream(buffer);
            case GZIP: // 使用GZIP压缩方式
                return new DataOutputStream(new GZIPOutputStream(buffer,
bufferSize));
            case SNAPPY: // 使用SNAPPY压缩方式
                try {
                    OutputStream stream = (OutputStream)
snappyOutputStreamSupplier.get()

```



```

        .newInstance(buffer, bufferSize); // 使用反射方式创建
        return new DataOutputStream(stream);
    } catch (Exception e) {
        throw new KafkaException(e);
    }
    case LZ4: // 使用 LZ4 压缩方式
        ... .. // 逻辑同 SNAPPY 压缩方式
    default: // 不支持的压缩方式, 抛出异常
        throw new IllegalArgumentException("Unknown compression
type: " + type);
    }
} catch (IOException e) {
    throw new KafkaException(e);
}
}

```

细心的读者可能会问, 为什么 GZIP 压缩方式会直接使用 new 创建, 而 Snappy 则使用反射方式呢? 这主要是因为 GZIP 使用的 GZIPOutputStream 是 JDK 自带的包, 而 Snappy 则需要引入额外的依赖包, 为了不引入 Snappy 压缩方式时, 减少依赖包, 这里使用反射的方式动态创建。这种设计的小技巧, 值得读者积累。在 Compressor 中还提供了 wrapForInput() 方法, 用于创建解压缩输入流, 逻辑与 wrapForOutput() 类似, 不再赘述。

Compressor 提供了一系列 put*() 方法, 向 appendStream 流写入数据, 如图 2-12 所示。很明显, 这是装饰器模式的典型, 通过 bufferStream 装饰, 添加自动扩容的功能; 通过 appendStream 装饰后, 添加压缩功能。

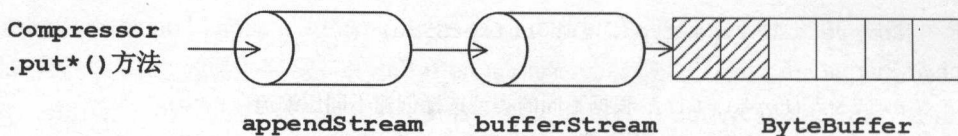


图 2-12

Compressor.estimateBytesWritten() 方法的功能是根据指定压缩方式的压缩率、写入的未压缩数据的字节数 (writtenUncompressed 字段记录)、估算因子 (COMPRESSION_RATE_ESTIMATION_FACTOR 字段), 估计已写入的 (压缩后的) 字节数, 此方法主要用在判断 MemoryRecords 是否写满的逻辑中使用。

上面对 Compressor 的分析比较简略, 为了便于读者理解, 下面的分析过程暂且认为

使用的是非压缩方式,在第4章对 Log SubSystem 中的 ByteBufferMessageSet 进行分析时,还会提及 Compressor。

了解了 Compressor 的实现逻辑之后,我们回到 MemoryRecords 继续分析。MemoryRecords 的构造方法是私有的,只能通过 emptyRecords() 方法得到其对象。MemoryRecords 中有四个比较重要的方法。

- append() 方法: 先判断 MemoryRecords 是否为可写模式,然后调用 Compressor.put*() 方法,将消息数据写入 ByteBuffer 中。
- hasRoomFor() 方法: 根据 Compressor 估算的已写字节数,估计 MemoryRecords 剩余空间是否足够写入指定的数据。注意,这里仅仅是估算,所以不一定准确,通过 hasRoomFor() 方法判断之后写入数据,也可能就会导致底层 ByteBuffer 出现扩容的情况。
- close() 方法: 出现 ByteBuffer 扩容的情况时,MemoryRecords.buffer 字段与 ByteBufferOutputStream.buffer 字段所指向的不再是同一个 ByteBuffer 对象,如图 2-13 (左) 所示。
- 在 close() 方法中,会将 MemoryRecords.buffer 字段指向扩容后的 ByteBuffer 对象,如图 2-13 (右) 所示。同时,将 writable 设置为 false (即只读模式)。

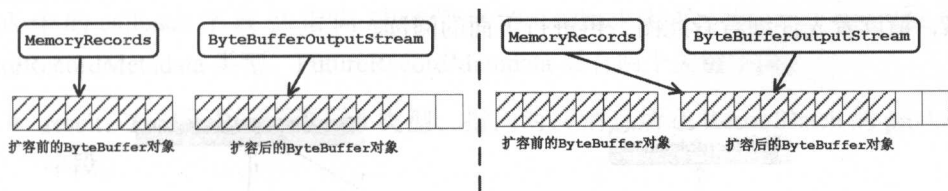


图 2-13

- sizeInBytes() 方法: 对于可写的 MemoryRecords, 返回的是 ByteBufferOutputStream.buffer 字段的大小; 对于只读 MemoryRecords, 返回的是 MemoryRecords.buffer 的大小。

MemoryRecords 还提供了迭代器, 主要是用在 Consumer 端读取其中的消息, 在第4章对 Kafka 服务端的 Log SubSystem 进行分析时, 再进行详细的分析。

2.3.2 RecordBatch

了解了 MemoryRecords 的具体实现之后, 来分析 RecordBatch 类的实现。如前面图 2-10

所示，每个 `RecordBatch` 对象中封装了一个 `MemoryRecords` 对象，除此之外，还封装了很多控制信息和统计信息，下面简单介绍一下。

- `recordCount`: 记录了保存的 `Record` 的个数。
- `maxRecordSize`: 最大 `Record` 的字节数。
- `attempts`: 尝试发送当前 `RecordBatch` 的次数。
- `lastAttemptMs`: 最后一次尝试发送的时间戳。
- `records`: 指向用来存储数据的 `MemoryRecords` 对象。
- `topicPartition`: 当前 `RecordBatch` 中缓存的消息都会发送给此 `TopicPartition`。
- `produceFuture`: `ProduceRequestResult` 类型，标识 `RecordBatch` 状态的 `Future` 对象。
- `lastAppendTime`: 最后一次向 `RecordBatch` 追加消息的时间戳。
- `thunks`: `Thunk` 对象的集合，在后面会详细介绍。
- `offsetCounter`: 用来记录某消息在 `RecordBatch` 中的偏移量。
- `retry`: 是否正在重试。如果 `RecordBatch` 中的数据发送失败，则会重新尝试发送。

图 2-14 中，以 `RecordBatch` 为中心，刻画了其相关类间的对应关系。为了防止读者感到困惑，请读者先仔细看看此图，再进行下面的阅读。

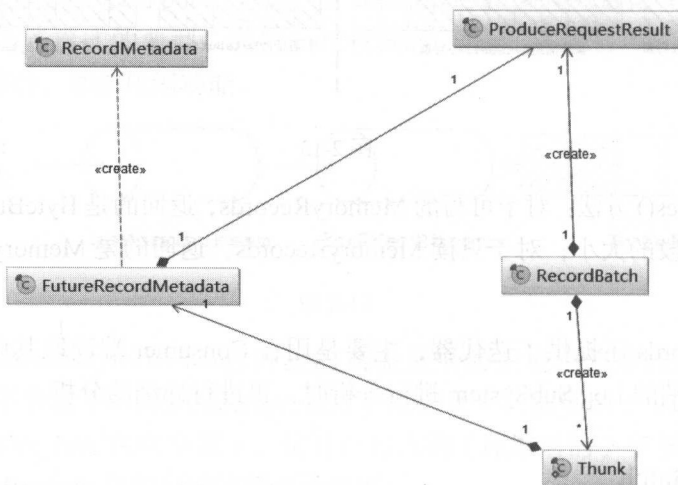


图 2-14

下面分析一下 `ProduceRequestResult` 这个类的功能。`ProduceRequestResult` 并未实现 `java.util.concurrent.Future` 接口，但是其通过包含一个 `count` 值为 1 的 `CountDownLatch` 对象，实现了类似于 `Future` 的功能（`Future`、`CountDownLatch` 等工具的使用，这里不做详细介绍，请读者参考相关资料）。

当 `RecordBatch` 中全部的消息被正常响应、或超时、或关闭生产者时，会调用 `ProduceRequestResult.done()` 方法，将 `produceFuture` 标记为完成并通过 `ProduceRequestResult.error` 字段区分“异常完成”还是“正常完成”，之后调用 `CountDownLatch` 对象的 `countDown()` 方法。此时，会唤醒阻塞在 `CountDownLatch` 对象的 `await()` 方法的线程（这些线程通过 `ProduceRequestResult` 的 `await` 方法等待上述三个事件的发生）。

在第 1 章介绍 Kafka 基本概念时提到过，分区会为其中记录的消息分配一个 `offset` 并通过此 `offset` 维护消息顺序。在 `ProduceRequestResult` 中还有一个需要注意的字段 `baseOffset`，表示的是服务端为此 `RecordBatch` 中第一条消息分配的 `offset`，这样每个消息可以根据此 `offset` 以及自身在此 `RecordBatch` 中的相对偏移量，计算出其在服务端分区中的偏移量了。

在介绍 `Tunk` 类之前，请读者回顾 `KafkaProducer.send()` 方法的第二个参数，是一个 `Callback` 对象，它是针对单个消息的回调函数（每个消息都会有一个对应的 `Callback` 对象作为回调）。`RecordBatch.thunks` 字段可以理解为消息的回调对象队列，`Thunk` 中的 `callback` 字段就指向对应消息的 `Callback` 对象，其另一个字段 `future` 是 `FutureRecordMetadata` 类型。`FutureRecordMetadata` 类有两个关键字段。

- `result`: `ProduceRequestResult` 类型，指向对应消息所在 `RecordBatch` 的 `produceFuture` 字段。
- `relativeOffset`: `long` 类型，记录了对应消息在 `RecordBatch` 中的偏移量。

`FutureRecordMetadata` 实现了 `java.util.concurrent.Future` 接口，但其实现基本都是委托给了 `ProduceRequestResult` 对应的方法，由此可以看出，消息应该是按照 `RecordBatch` 进行发送和确认的，在后面的分析中还会介绍。

当生产者已经收到某消息的响应时，`FutureRecordMetadata.get()` 方法就会返回 `RecordMetadata` 对象，其中包含消息在 `Partition` 中的 `offset` 等其他元数据，可供用户自定义 `Callback` 使用。

分析完 `RecordBatch` 依赖的组件，现在回来看看 `RecordBatch` 类的核心方法。`tryAppend()` 方法是最核心的方法，其功能是尝试将消息添加到当前的 `RecordBatch` 中缓存，代码如下所示。

```

public FutureRecordMetadata tryAppend(long timestamp, byte[] key,
    byte[] value, Callback callback, long now) {
    // 估算剩余空间不足, 前面说过, 这不是一个准确值
    if (!this.records.hasRoomFor(key, value)) {
        return null;
    } else {
        // 向 MemoryRecords 中添加数据。注意, offsetCounter 是在 RecordBatch 中的偏移量
        long checksum = this.records.append(offsetCounter++, timestamp, key,
            value);
        // 更新统计信息 (省略)
        // 创建 FutureRecordMetadata 对象
        FutureRecordMetadata future = new FutureRecordMetadata(
            this.produceFuture, this.recordCount, timestamp, checksum,
            key == null ? -1 : key.length, value == null ? -1 :
            value.length);
        // 将用户自定义 CallBack 和 FutureRecordMetadata 封装成 Thunk, 保存到 thunks 集合中
        if (callback != null) thunks.add(new Thunk(callback, future));
        this.recordCount++; // 更新 recordCount
        return future; // 返回 FutureRecordMetadata 对象
    }
}

```

当 RecordBatch 成功收到正常响应、或超时、或关闭生产者时, 都会调用 RecordBatch 的 done() 方法。在 done() 方法中, 会回调 RecordBatch 中全部消息的 Callback 回调, 并调用其 produceFuture 字段的 done() 方法。RecordBatch.done() 方法的调用关系如图 2-15 所示。

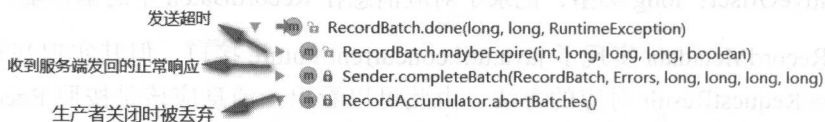


图 2-15

RecordBatch.done() 方法的代码如下:


```

public void done(long baseOffset, long timestamp, RuntimeException
exception) {
    for (int i = 0; i < this.thunks.size(); i++) { // 循环执行每个消息的
Callback
        try {
            Thunk thunk = this.thunks.get(i);
            if (exception == null) { // 正常处理完成
                // 将服务端返回的信息 (offset 和 timestamp) 和消息的其他信息封装成 RecordMetadata
                RecordMetadata metadata = new RecordMetadata(
                    this.topicPartition, baseOffset, thunk.future.
relativeOffset(),
                    timestamp == Record.NO_TIMESTAMP ? thunk.future.
timestamp() :
                        timestamp, thunk.future.checksum(),
                        thunk.future.serializedKeySize(),
                        thunk.future.serializedValueSize());
                // 调用消息对应的自定义 Callback
                thunk.callback.onCompletion(metadata, null);
                // 处理过程中出现异常的情况, 注意, 第一个参数为 null, 与上面正常的情况相反
            } else {
                unk.callback.onCompletion(null, exception);
            }
        } catch (Exception e) {
            ... ..
        }
    }
    // 标识整个 RecordBatch 都已经处理完成
    this.produceFuture.done(topicPartition, baseOffset, exception);
}

```

maybeExpire 方法调用 done 方法并抛出 TimeoutException, 标识整个 RecordBatch 中的消息过期, 比较简单, 不再赘述。

2.3.3 BufferPool

ByteBuffer 的创建和释放是比较消耗资源的, 为了实现内存的高效利用, 基本上每个成熟的框架或工具都有一套内存管理机制。Kafka 客户端使用 BufferPool 来实现 ByteBuffer 的复用。图 2-16 展示了 BufferPool 的核心字段。

BufferPool	
totalMemory	long
poolableSize	int
lock	ReentrantLock
free	Deque<ByteBuffer>
waiters	Deque<Condition>
availableMemory	long

图 2-16

首先需要了解的是，每个 BufferPool 对象只针对特定大小（由 poolableSize 字段指定）的 ByteBuffer 进行管理，对于其他大小的 ByteBuffer 并不会缓存进 BufferPool。一般情况下，我们会调整 MemoryRecords 的大小（RecordAccumulator.batchSize 字段指定），使每个 MemoryRecords 可以缓存多条消息。但也有例外情况，当一条消息的字节数大于 MemoryRecords 时，就不会复用 BufferPool 中缓存的 ByteBuffer，而是额外分配 ByteBuffer，在它被使用完后也不会放入 BufferPool 进行管理，而是直接丢弃由 GC 回收。如果经常出现这种例外情况，就需要考虑调整 batchSize 的配置了。

下面介绍 BufferPool 的关键字段：

- free：是一个 ArrayDeque<ByteBuffer> 队列，其中缓存了指定大小的 ByteBuffer 对象。
- ReentrantLock：因为有多线程并发分配和回收 ByteBuffer，所以使用锁控制并发，保证线程安全。
- waiters：记录因申请不到足够空间而阻塞的线程，此队列中实际记录的是阻塞线程对应的 Condition 对象。
- totalMemory：记录了整个 Pool 的大小。
- availableMemory：记录了可用的空间大小，这个空间是 totalMemory 减去 free 列表中全部 ByteBuffer 的大小。

BufferPool.allocate() 方法负责从缓冲池中申请 ByteBuffer，当缓冲池中空间不足时，就会阻塞调用线程。下面简单分析一下 allocate() 方法申请空间的过程：


```

public ByteBuffer allocate(int size, long maxTimeToBlockMs) {
    if (size > this.totalMemory) throw new IllegalArgumentException("...");
    this.lock.lock(); // 加锁同步
    try {
        // 请求的是 poolableSize 指定大小的 ByteBuffer, 且 free 中有空闲的 ByteBuffer
        if (size == poolableSize && !this.free.isEmpty())
            return this.free.pollFirst(); // 返回合适的 ByteBuffer
        // 当申请的空间大小不是 poolableSize, 则执行下面的处理
        // free 队列中都是 poolableSize 大小的 ByteBuffer, 可以直接计算整个 free 队列的空间
        int freeListSize = this.free.size() * this.poolableSize;
        if (this.availableMemory + freeListSize >= size) {
            // 为了让 availableMemory > size, freeUp() 方法会从 free 队列中不断释放
            // ByteBuffer, 直到 availableMemory 满足这次申请
            freeUp(size);
            this.availableMemory -= size; // 减小 availableMemory
            lock.unlock(); // 解锁
        }
        // 这里并没有使用 free 队列中的 buffer, 而是直接分配 size 大小的 HeapByteBuffer
        return ByteBuffer.allocate(size);
    } else { // 没有足够空间, 只能阻塞了
        int accumulated = 0;
        ByteBuffer buffer = null;
        Condition moreMemory = this.lock.newCondition();
        this.waiters.addLast(moreMemory); // 将 Condition 添加到 waiters 中
        while (accumulated < size) { // 循环等待
            long startWaitNs = time.nanoseconds();
            long timeNs;
            boolean waitingTimeElapsed;
            try {
                waitingTimeElapsed = !moreMemory.await(
                    remainingTimeToBlockNs, TimeUnit.NANOSECONDS); // 阻塞
            } catch (InterruptedException e) {
                // 异常, 移除此线程对应的 Condition
                this.waiters.remove(moreMemory);
                throw e;
            } finally {

```

```

    ... ..
    // 统计阻塞时间
    this.waitTime.record(timeNs, time.milliseconds());
}
if (waitingTimeElapsed) { // 超时, 报错
    this.waiters.remove(moreMemory);
    throw new TimeoutException("...");
}

remainingTimeToBlockNs -= timeNs;
// 请求的是 poolableSize 大小的 ByteBuffer, 且 free 中有空闲的 ByteBuffer
if (accumulated == 0 && size == this.poolableSize
    && !this.free.isEmpty()) {
    buffer = this.free.pollFirst();
    accumulated = size;
} else { // 先分配一部分空间, 并继续等待空闲空间
    freeUp(size - accumulated);
    int got = (int) Math.min(size - accumulated,
        this.availableMemory);
    this.availableMemory -= got;
    accumulated += got;
}

}

// 已经成功分配空间, 移除 Condition
Condition removed = this.waiters.removeFirst();
// 要是还有空闲空间, 就唤醒下一个线程
if (this.availableMemory > 0 || !this.free.isEmpty()) {
    if (!this.waiters.isEmpty())
        this.waiters.peekFirst().signal();
}
lock.unlock(); // 解锁
}
} finally { // 解锁 (略) }
}

```

了解了 `allocate()` 方法的实现后, 继续分析 `deallocate()` 方法的实现:

```

public void deallocate(ByteBuffer buffer, int size) {
    lock.lock(); // 加锁 ...
    try {
        // 释放的 ByteBuffer 的大小是 poolableSize, 放入 free 队列中进行管理
        if (size == this.poolableSize && size == buffer.capacity()) {
            buffer.clear();
            this.free.add(buffer);
        } else {
            // 释放的 ByteBuffer 大小不是 poolableSize, 不会复用 ByteBuffer, 仅修改
            // availableMemory 的值
            this.availableMemory += size;
        }
        // 唤醒一个因空间不足而阻塞的线程
        Condition moreMem = this.waiters.peekFirst();
        if (moreMem != null)
            moreMem.signal();
    } finally {
        lock.unlock(); // 解锁
    }
}

```

2.3.4 RecordAccumulator

介绍完了 MemoryRecord、RecordBatch 以及 BufferPool 的工作机制，再来看 RecordAccumulator 的实现就比较简单了。下面来看 RecordAccumulator 中的关键字段和方法，如图 2-17 所示。

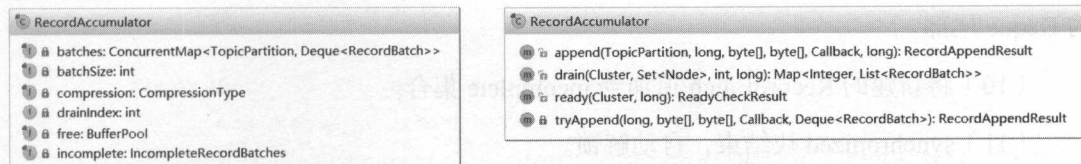


图 2-17

- **batches**: TopicPartition 与 RecordBatch 集合的映射关系，类型是 CopyOnWriteMap，是线程安全的集合，但其中的 Deque 是 ArrayDeque 类型，是非线程安全的集合。在后面的介绍中可以看到，追加新消息或发送 RecordBatch 的时候，需要加锁同步。

每个 Deque 中都保存了发往对应 TopicPartition 的 RecordBatch 集合。

- batchSize: 指定每个 RecordBatch 底层 ByteBuffer 的大小。
- Compression: 压缩类型, 参考 MemoryRecords 小节。
- incomplete: 未发送完成的 RecordBatch 集合, 底层通过 Set<RecordBatch> 集合实现。
- free: BufferPool 对象, 参考 BufferPool 小节。
- drainIndex: 使用 drain 方法批量导出 RecordBatch 时, 为了防止饥饿, 使用 drainIndex 记录上次发送停止时的位置, 下次继续从此位置开始发送。

KafkaProducer.send() 方法最终会调用 RecordsAccumulator.append() 方法将消息追加到 RecordAccumulator 中, 其代码比较长, 先来看其主要逻辑:

(1) 首先在 batches 集合中查找 TopicPartition 对应的 Deque, 查找不到, 则创建新的 Deque, 并添加到 batches 集合中。

(2) 对 Deque 加锁 (使用 synchronized 关键字加锁)。

(3) 调用 tryAppend() 方法, 尝试向 Deque 中最后一个 RecordBatch 追加 Record。

(4) synchronized 块结束, 自动解锁。

(5) 追加成功, 则返回 RecordAppendResult (其中封装了 ProduceRequestResult)。

(6) 追加失败, 则尝试从 BufferPool 中申请新的 ByteBuffer。

(7) 对 Deque 加锁 (使用 synchronized 关键字加锁), 再次尝试第 3 步。

(8) 追加成功, 则返回; 失败, 则使用第 5 步得到的 ByteBuffer 创建 RecordBatch。

(9) 将 Record 追加到新建的 RecordBatch 中, 并将新建的 RecordBatch 追加到对应的 Deque 尾部。

(10) 将新建的 RecordBatch 追加到 incomplete 集合。

(11) synchronized 块结束, 自动解锁。

(12) 返回 RecordAppendResult, RecordAppendResult 会中的字段会作为唤醒 Sender 线程的条件。

下面是 RecordsAccumulator.append() 方法的具体实现:


```

public RecordAppendResult append(TopicPartition tp, long timestamp, byte[]
key,
    byte[] value, Callback callback,
    long maxTimeToBlock) throws InterruptedException {
    // 统计正在向 RecordsAccumulator 中追加数据的线程数
    appendsInProgress.incrementAndGet();
    try {
        // 步骤 1: 查找 TopicPartition 对应的 Deque
        Deque<RecordBatch> dq = getOrCreateDeque(tp);
        synchronized (dq) { // 步骤 2: 对 Deque 对象加锁
            ... .. // 边界检查 (略)
            // 步骤 3: 向 Deque 中最后一个 RecordBatch 追加 Record
            RecordAppendResult appendResult = tryAppend(timestamp, key,
                value, callback, dq);
            if (appendResult != null)
                return appendResult; // 步骤 5: 追加成功则直接返回
        } // 步骤 4: synchronized 块结束, 解锁

        // 步骤 6: 追加失败, 从 BufferPool 中申请新空间
        ByteBuffer buffer = free.allocate(size, maxTimeToBlock);
        synchronized (dq) {
            ... .. // 边界检查 (略)
            // 步骤 7: 对 Deque 加锁后, 再次调用 tryAppend() 方法尝试追加 Record
            RecordAppendResult appendResult = tryAppend(timestamp, key,
                value, callback, dq);
            if (appendResult != null) { // 步骤 8: 追加成功, 则返回
                free.deallocate(buffer); // 释放步骤 7 中申请的新空间
                return appendResult;
            }
            MemoryRecords records = MemoryRecords.emptyRecords(buffer,
                compression, this.batchSize);
            RecordBatch batch = new RecordBatch(tp, records, time.
milliseconds());
            // 步骤 9: 在新创建的 RecordBatch 中追加 Record, 并将其添加到 batches 集合中
            FutureRecordMetadata future = Utils.notNull(batch.tryAppend(
                timestamp, key, value, callback, time.milliseconds()));
            dq.addLast(batch);

```

```

// 步骤 10: 将新建的 RecordBatch 追加到 incomplete 集合
incomplete.add(batch);
return new RecordAppendResult(future, dq.size() > 1
    // 步骤 12: 返回 RecordAppendResult
    || batch.records.isFull(), true);
} // 步骤 11: synchronized 块结束, 解锁
} finally {
    appendsInProgress.decrementAndGet();
}
}

```

`RecordsAccumulator.tryAppend()` 方法会查找 `batches` 集合中对应队列的最后一个 `RecordBatch` 对象, 并调用其 `tryAppend()` 方法完成消息追加。代码比较简单, 不再贴出来了。

这里需要注意, 在第 2、6 步对 `Deque` 加 `synchronized` 锁然后重试的原因。这里的 `Deque` 使用的是 `ArrayDeque` (非线程安全), 所以需要加锁同步。有读者可能会问, 那为什么分多个 `synchronized` 块而不是在一个完整的 `synchronized` 块中完成呢? 主要是因为向 `BufferPool` 申请新 `ByteBuffer` 的时候, 可能会导致阻塞。我们假设在一个 `synchronized` 块中完成上面所有追加操作, 有下面的场景: 线程 1 发送的消息比较大, 需要向 `BufferPool` 申请新空间, 而此时 `BufferPool` 空间不足, 线程 1 在 `BufferPool` 上等待, 此时它依然持有对应 `Deque` 的锁; 线程 2 发送的消息较小, `Deque` 最后一个 `RecordBatch` 剩余空间够用, 但是由于线程 1 未释放 `Deque` 的锁, 所以也需要一起等待。若线程 2 这样的线程较多, 就会造成很多不必要的线程阻塞, 降低了吞吐量。这里体现了“减少锁的持有时间”这一优化手段, 值得读者积累。

第二次加锁后重试, 是为了防止多个线程并发向 `BufferPool` 申请空间后, 造成内部碎片。这种场景如图 2-18 所示, 线程 1 发现最后一个 `RecordBatch` 空间不够用, 申请空间并创建一个新 `RecordBatch` 对象添加到 `Deque` 的尾部; 线程 2 与线程 1 并发执行, 也将新创建一个 `RecordBatch` 添加到 `Deque` 尾部。从上面的逻辑中我们可以得知, 之后的追加操作只会在 `Deque` 尾部进行, 这样就会出现下图的场景, `RecordBatch4` 不再被使用, 这就出现了内部碎片。

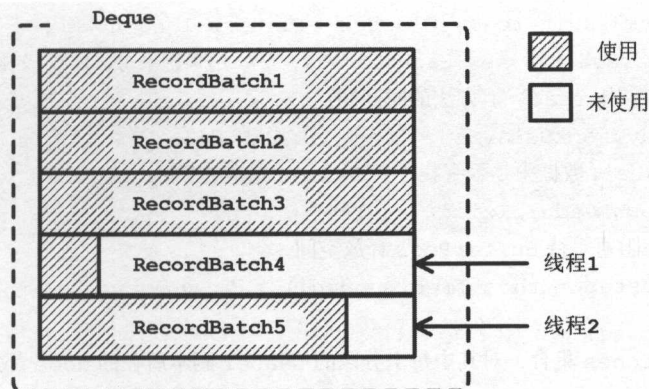


图 2-18

现在回到 `KafkaProducer.doSend()` 方法，`doSend()` 方法的最后一步就是判断此次向 `RecordAccumulator` 中追加消息后是否满足唤醒 `Sender` 线程条件，这里唤醒 `Sender` 线程的条件是消息所在队列的最后一个 `RecordBatch` 满了或此队列中不止一个 `RecordBatch`。

在客户端将消息发送给服务端之前，会调用 `RecordAccumulator.ready()` 方法获取集群中符合发送消息条件的节点集合。这些条件是站在 `RecordAccumulator` 的角度对集群中的 `Node` 进行筛选的，具体的条件如下：

- (1) `Deque` 中有多个 `RecordBatch` 或是第一个 `RecordBatch` 是否满了。
- (2) 是否超时了。
- (3) 是否有其他线程在等待 `BufferPool` 释放空间（即 `BufferPool` 的空间耗尽了）。
- (4) 是否有线程正在等待 `flush` 操作完成。
- (5) `Sender` 线程准备关闭。

下面来看一下 `ready` 方法的代码，它会遍历 `batches` 集合中每个分区，首先查找当前分区 `Leader` 副本所在的 `Node`，如果满足上述五个条件，则将此 `Node` 信息记录到 `readyNodes` 集合中。遍历完成后返回 `ReadyCheckResult` 对象，其中记录了满足发送条件的 `Node` 集合、在遍历过程中是否有找不到 `Leader` 副本的分区（也可以认为是 `Metadata` 中当前的元数据过时了）、下次调用 `ready()` 方法进行检查的时间间隔。

```

public ReadyCheckResult ready(Cluster cluster, long nowMs) {
    Set<Node> readyNodes = new HashSet<>(); // 用来记录可以向哪些 Node 节点发送信息
    // 记录下次需要调用 ready() 方法的时间间隔
    long nextReadyCheckDelayMs = Long.MAX_VALUE;
    // 根据 Metadata 元数据中是否有找不到 Leader 副本的分区
    boolean unknownLeadersExist = false;
    // 是否有线程在阻塞等待 BufferPool 释放空间
    boolean exhausted = this.free.queued() > 0;

    // 下面遍历 batches 集合, 对其中每个分区的 Leader 副本所在的 Node 都进行判断
    for (Map.Entry<TopicPartition, Deque<RecordBatch>> entry : this.batches.
entrySet()) {
        ... ..
        Node leader = cluster.leaderFor(part); // 查找分区的 Leader 副本所在的 Node
        // 根据 Cluster 的信息检查 Leader, Leader 找不到, 肯定不能发送消息
        if (leader == null) {
            unknownLeadersExist = true; // 标记为 true, 之后会触发 Metadata 的更新
        } else if (!readyNodes.contains(leader) && !muted.contains(part)) {
            synchronized (deque) { // 加锁读取 Deque 中的元素
                // 只取 Deque 中的第一个 RecordBatch
                RecordBatch batch = deque.peekFirst();
                if (batch != null) {
                    // 这里通过计算得到下面五个条件, 具体计算过程略(略)
                    boolean full = deque.size() > 1 || batch.records.
isFull(); // 条件 1
                    boolean expired = waitedTimeMs >= timeToWaitMs; // 条件 2
                    boolean sendable = full || expired || exhausted // 条件 3
                        || flushInProgress() // 条件 4
                        || closed // 条件 5
                    if (sendable && !backingOff)
                        readyNodes.add(leader);
                    else
                        // 记录下次需要调用 ready() 方法检查的时间间隔
                        nextReadyCheckDelayMs = Math.min(timeLeftMs,
nextReadyCheckDelayMs);
                }
            }
        }
    }
}

```

```

    }
}
return new ReadyCheckResult(readyNodes, nextReadyCheckDelayMs,
    unknownLeadersExist);
}

```

调用 `RecordAccumulator.ready()` 方法得到 `readyNodes` 集合后，此集合还要经过 `NetworkClient` 的过滤（在介绍 `Sender` 线程的时候再详细介绍）之后，才能得到最终能够发送消息的 `Node` 集合。

`RecordAccumulator.drain()` 方法会根据上述 `Node` 集合获取要发送的消息，返回 `Map<Integer, List<RecordBatch>>` 集合，`key` 是 `NodeId`，`value` 是待发送的 `RecordBatch` 集合。`drain` 方法也是由 `Sender` 线程调用的。`drain()` 方法的核心逻辑是进行映射的转换：将 `RecordAccumulator` 记录的 `TopicPartition->RecordBatch` 集合的映射，转换成了 `NodeId->RecordBatch` 集合的映射。为什么需要这次转换呢？在网络 I/O 层面，生产者是面向 `Node` 节点发送消息数据，它只建立到 `Node` 的连接并发送数据，并不关心这些数据属于哪个 `TopicPartition`；而在调用 `KafkaProducer` 的上层业务逻辑中，则是按照 `TopicPartition` 的方式产生数据，它只关心发送到哪个 `TopicPartition`，并不关心这些 `TopicPartition` 在哪个 `Node` 节点上。在下文介绍到 `Sender` 线程的时候会发现，它每次向每个 `Node` 节点至多发送一个 `ClientRequest` 请求，其中封装了追加到此 `Node` 节点上多个分区的信息，待请求到达服务端后，由 `Kafka` 对请求记性解析。下面来看看 `drain` 方法的代码：

```

public Map<Integer, List<RecordBatch>> drain(Cluster cluster,
    Set<Node> nodes, int maxSize, long now) {
    // 转换后的结果
    Map<Integer, List<RecordBatch>> batches = new HashMap<>();
    for (Node node : nodes) { // 遍历指定的 ready Node 集合
        int size = 0;
        // 获取当前 Node 上的分区集合
        List<PartitionInfo> parts = cluster.partitionsForNode(node.id());
        List<RecordBatch> ready = new ArrayList<>(); // 记录要发送的 RecordBatch
        // drainIndex 是 batches 的下标，记录上次发送停止时的位置，下次继续从此位置开始发送
        // 若一直从索引 0 的队列开始发送，可能会出现一直只发送前几个分区的情况，造成其他
        // 分区饥饿
    }
}

```



```

int start = drainIndex = drainIndex % parts.size();
do {
    PartitionInfo part = parts.get(drainIndex); // 获取分区的详细情况
    TopicPartition tp = new TopicPartition(part.topic(), part.
partition());
    Deque<RecordBatch> deque = getDeque(tp); // 获取对应的 RecordBatch 队列
    ... ..// 此处省略部分边界检查代码, 感兴趣的读者请参考源代码
    synchronized (deque) {
        RecordBatch first = deque.peekFirst(); // 获取队列中第一个 RecordBatch
        if (size + first.records.sizeInBytes() > maxSize && !ready.
isEmpty()) {
            break; // 数据量已满, 结束循环, 一般是一个请求的大小
        } else {
            // 从队列中获取一个 RecordBatch, 并将这个 RecordBatch 放到 ready 集合中
            // 每个 TopicPartition 只取一个 RecordBatch
            RecordBatch batch = deque.pollFirst();
            // 关闭 Compressor 及底层输出流, 并将 MemoryRecords 设置为只读
            batch.records.close();
            size += batch.records.sizeInBytes();
            ready.add(batch);
            batch.drainedMs = now;
        }
    }
    // 更新 drainIndex
    this.drainIndex = (this.drainIndex + 1) % parts.size();
} while (start != drainIndex);
batches.put(node.id(), ready); // 记录 NodeId 与 RecordBatch 的对应关系
}
return batches;
}

```

注意, 在上面代码中, 只从每个队列中取出一个 RecordBatch 放到 ready 集合中, 这也是为了防止饥饿, 提高系统的可用性。

RecordAccumulator 的工作原理到这里就介绍完了, 整个 KafkaProducer.send() 方法过程中用到的所有组件也都分析完了。下一节, 我们分析 Sender 线程是如何发送消息的。

2.4 Sender 分析

通过前面的分析我们知道，主线程通过 `KafkaProducer.send()` 方法将消息放入 `RecordAccumulator` 中缓存，并没有实际的网络 I/O 操作。网络 I/O 操作是由 `Sender` 线程统一进行的。

我们先来了解一下 `Sender` 线程发送消息的整个流程：首先，它根据 `RecordAccumulator` 的缓存情况，筛选出可以向哪些 `Node` 节点发送消息，即上一节介绍的 `RecordAccumulator.ready()` 方法；然后，根据生产者与各个节点的连接情况（由 `NetworkClient` 管理），过滤 `Node` 节点；之后，生成相应的请求，这里要特别注意的是，每个 `Node` 节点只生成一个请求；最后，调用 `NetworkClient` 将请求发送出去。图 2-19 展示了 `Sender` 依赖的三个比较关键的组件。

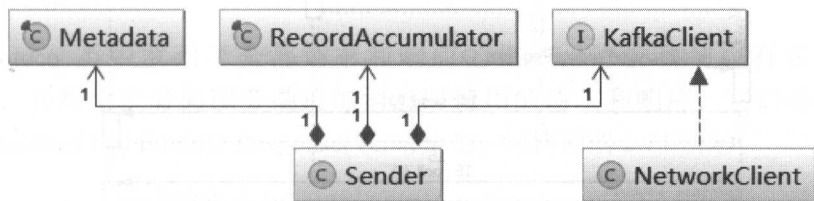


图 2-19

`Sender` 实现了 `Runnable` 接口，并运行在单独的 `ioThread` 中。`Sender` 的 `run()` 方法调用了其重载 `run(long)`，这才是 `Sender` 线程的核心方法，也是发送消息的关键流程，其时序图如图 2-20 所示。

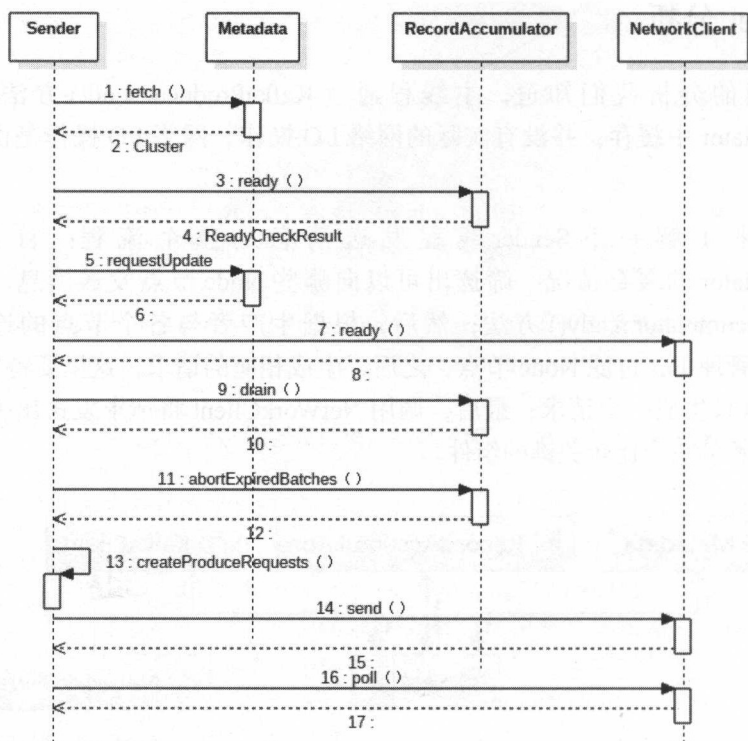


图 2-20

下面简述 run(long) 方法的流程：

- (1) 从 Metadata 获取 Kafka 集群元数据。
- (2) 调用 RecordAccumulator.ready() 方法，根据 RecordAccumulator 的缓存情况，选出可以向哪些 Node 节点发送消息，返回 ReadyCheckResult 对象。
- (3) 如果 ReadyCheckResult 中标识有 unknownLeadersExist，则调用 Metadata 的 requestUpdate 方法，标记需要更新 Kafka 的集群信息。
- (4) 针对 ReadyCheckResult 中 readyNodes 集合，循环调用 NetworkClient.ready() 方法，目的是检查网络 I/O 方面是否符合发送消息的条件，不符合条件的 Node 将会从 readyNodes 集合中删除。NetworkClient 类的具体实现在后面详细介绍。
- (5) 针对经过步骤 4 处理后的 readyNodes 集合，调用 RecordAccumulator.drain() 方法，获取待发送的消息集合。
- (6) 调用 RecordAccumulator.abortExpiredBatches() 方法处理 RecordAccumulator 中

超时的消息。其代码逻辑是，遍历 RecordAccumulator 中保存的全部 RecordBatch，调用 RecordBatch.maybeExpire() 方法进行处理。如果已超时，则调用 RecordBatch.done() 方法，其中会触发自定义 Callback，并将 RecordBatch 从队列中移除，释放 ByteBuffer 空间。

(7) 调用 Sender.createProduceRequests() 方法将待发送的消息封装成 ClientRequest，此方法后面会详细描述。

(8) 调用 NetWorkClient.send() 方法，将 ClientRequest 写入 KafkaChannel 的 send 字段。

(9) 调用 NetWorkClient.poll() 方法，将 KafkaChannel.send 字段中保存的 ClientRequest 发送出去，同时，还会处理服务端发回的响应、处理超时的请求、调用用户自定义 Callback 等。

在本节，我们将详细介绍上述步骤，并分析依赖的组件的原理和实现。

2.4.1 创建请求

在 Protocol 类中罗列了全部请求和响应的格式，请求和响应有多个不同的版本。首先，介绍生产者向服务端追加消息时使用的请求和响应，它们分别是 Produce Request(Version:2) 和 Produce Response(Version: 2)，结构如图 2-21 所示。

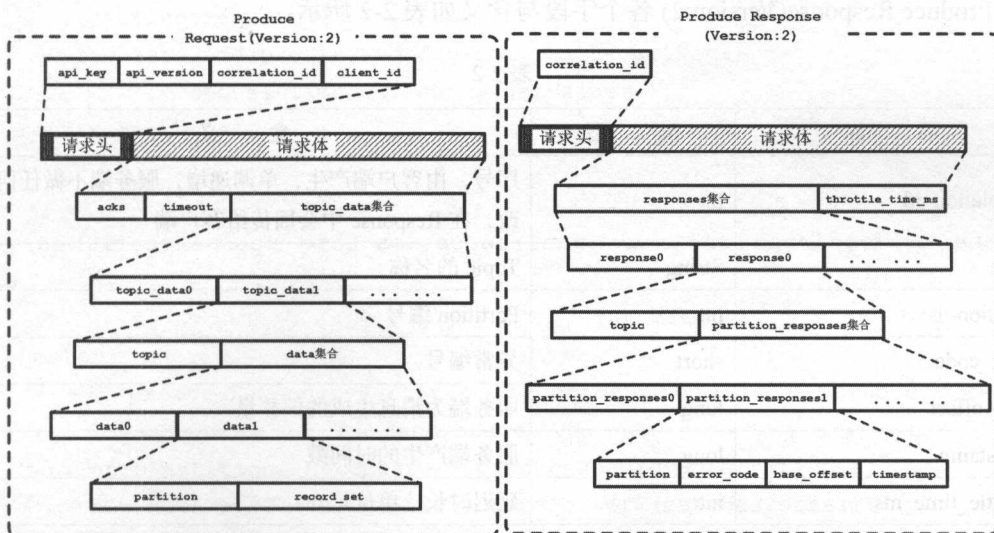


图 2-21

Produce Request(Version:2) 的请求头和请求体各个字段的含义如表 2-1 所示。

表 2-1

名 称	类 型	含 义
api_key	short	API 标识
api_version	short	API 版本号
correlation_id	int	序号，由客户端产生，单调递增，服务端不做任何修改，在 Response 中会回传给客户端
client_id	String	客户端 ID，可为 null
acks	short	指定服务端响应此请求之前，需要有多少 Replica 成功复制了此请求的消息。-1 表示整个 ISR 都完成了复制
timeout	int	超时时间，单位是 ms
topic	String	Topic 的名称
partition	int	Partition 编号
record_set	byte 数组	消息的有效负载

Produce Response(Version:2) 各个字段与含义如表 2-2 所示。

表 2-2

名 称	类 型	含 义
correlation_id	int	序号，由客户端产生，单调递增，服务端不做任何修改，在 Response 中会回传给客户端
topic	String	Topic 的名称
partition	int	Partition 编号
error_code	short	异常编号
base_offset	long	服务端为消息生成的偏移量
timestamp	long	服务端产生的时间戳
throttle_time_ms	int	延迟时长，单位是 ms

Sender.createProduceRequests() 方法的功能是将待发送的消息封装成 ClientRequest。不管一个 Node 对应有多少个 RecordBatch，也不管这些 RecordBatch 是发给几个分区的，每个 Node 至多生成一个 ClientRequest 对象。创建 ClientRequest 的核心逻辑如下：

(1) 将一个 NodeId 对应的 RecordBatch 集合，重新整理为 produceRecordsByPartition

(Map<TopicPartition, ByteBuffer>) 和 recordsByPartition (Map<TopicPartition, RecordBatch>) 两个集合。

(2) 创建 RequestSend, RequestSend 是真正通过网络 I/O 发送的对象, 其格式符合上面描述的 Produce Request (Version:2) 协议, 其中有效负载就是 produceRecordsByPartition 中的数据。

(3) 创建 RequestCompletionHandler 作为回调对象。

(4) 将 RequestSend 对象和 RequestCompletionHandler 对象封装进 ClientRequest 对象中, 并将其返回。

下面来看 Sender.createProduceRequests() 方法的具体实现:

```
private List<ClientRequest> createProduceRequests (
    Map<Integer, List<RecordBatch>> collated, long now) {
    List<ClientRequest> requests = new ArrayList<ClientRequest>(collated.
size());
    for (Map.Entry<Integer, List<RecordBatch>> entry : collated.entrySet())
        // 调用 produceRequest() 方法, 将发往同一 Node 的 RecordBatch 封装成一个
        // ClientRequest 对象
        requests.add(produceRequest(now, entry.getKey(), acks,
            requestTimeout, entry.getValue()));
    return requests;
}

private ClientRequest produceRequest(long now, int destination, short
acks,
    int timeout, List<RecordBatch>
batches) {
    // 注意: produceRecordsByPartition 和 recordsByPartition 的 value 是不一样的
    // 一个是 ByteBuffer。一个是 RecordBatch
    Map<TopicPartition, ByteBuffer> produceRecordsByPartition =
        new HashMap<TopicPartition, ByteBuffer>(batches.size());
    final Map<TopicPartition, RecordBatch> recordsByPartition =
        new HashMap<TopicPartition, RecordBatch>(batches.size());
    // 步骤 1: 将 RecordBatch 列表按照 partition 进行分类, 整理成上述两个集合
    for (RecordBatch batch : batches) {
        TopicPartition tp = batch.topicPartition;
```

```

        produceRecordsByPartition.put(tp, batch.records.buffer());
        recordsByPartition.put(tp, batch);
    }
    // 步骤 2: 创建 ProduceRequest 和 RequestSend
    ProduceRequest request = new ProduceRequest(acks, timeout,
        produceRecordsByPartition);
    RequestSend send = new RequestSend(Integer.toString(destination),
        this.client.nextRequestHeader(ApiKeys.PRODUCE), request.
toStruct());

    // 步骤 3: 创建 RequestCompletionHandler 作为回调对象, 其具体逻辑在后面会详细介绍
    RequestCompletionHandler callback = new RequestCompletionHandler() {
        public void onComplete(ClientResponse response) {
            handleProduceResponse(response, recordsByPartition,
                time.milliseconds());
        }
    };

    // 创建 ClientRequest 对象。注意其第二个参数, 根据 acks 配置决定请求是否需要获取响应
    return new ClientRequest(now, acks != 0, send, callback);
}

```

到这里, `ProduceRequest` 的格式以及创建过程就分析完了。创建在后面的流程中, 发送的是 `RequestSend` 对象, 会将 `ClientRequest` 放入 `InFlightRequest` 中缓存, 当请求收到响应或出现异常时, 通过缓存的 `ClientRequest` 调用其 `RequestCompletionHandler` 对象。具体处理过程在后面会详细介绍。

2.4.2 KSelector

在介绍 `NetworkClient` 之前, 我们先来了解 `NetworkClient` 的整个结构, 以及其依赖的其他组件, 如图 2-22 所示。

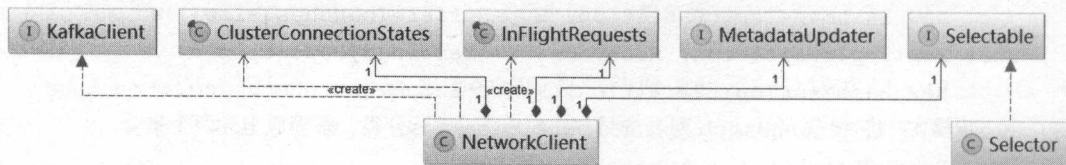


图 2-22

需要注意的是，图 2-22 中的 Selector 的类型并不是 `java.nio.channels.Selector`，而是 `org.apache.kafka.common.network.Selector`，为了方便区分和描述，将其简称为 KSelect。KSelector 使用 NIO 异步非阻塞模式实现网络 I/O 操作，KSelector 使用一个单独的线程可以管理多条网络连接上的连接、读、写等操作。下面介绍 KSelector 的核心字段和方法，如图 2-23 所示。

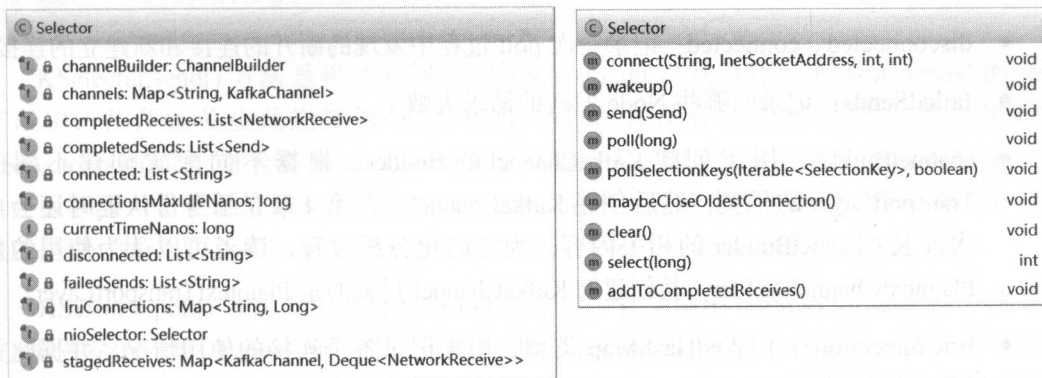


图 2-23

下面先介绍 KSelector 的字段。

- `nioSelector`: `java.nio.channels.Selector` 类型，用来监听网络 I/O 事件。
- `channels`: `HashMap<String, KafkaChannel>` 类型，维护了 `NodeId` 与 `KafkaChannel` 之间的映射关系，表示生产者客户端与各个 `Node` 之间的网络连接。`KafkaChannel` 是在 `SocketChannel` 上的又一层封装，如图 2-24 所示，其中 `Send` 和 `NetworkReceive` 分别表示读和写时用的缓存，底层通过 `ByteBuffer` 实现，`TransportLayer` 封装 `SocketChannel` 及 `SelectionKey`，`TransportLayer` 根据网络协议的不同，提供不同的子类，而对 `KafkaChannel` 提供统一的接口，这是策略模式很好的应用，请读者积累。

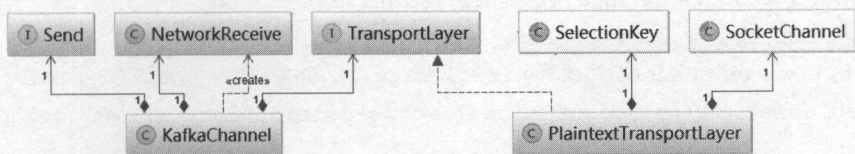


图 2-24

- `completedSends`: 记录已经完全发送出去请求。
- `completedReceives`: 记录已经完全接收到的请求。
- `stagedReceives`: 暂存一次 `OP_READ` 事件处理过程中读取到的全部请求。当一次 `OP_READ` 事件处理完成之后, 会将 `stagedReceives` 集合中的请求保存到 `completeReceives` 集合中。
- `disconnected`、`connected`: 记录一次 `poll` 过程中发现的断开的连接和新建立连接。
- `failedSends`: 记录向哪些 `Node` 发送的请求失败了。
- `channelBuilder`: 用于创建 `KafkaChannel` 的 `Builder`。根据不同配置创建不同的 `TransportLayer` 的子类, 然后创建 `KafkaChannel`。在第 4 章介绍身份认证时还会再次涉及 `ChannelBuilder` 的相关内容, 为了简化分析过程, 读者可以认为使用的是 `PlaintextChannelBuilder`, 其创建的 `KafkaChannel` 封装的是 `PlaintextTransportLayer`。
- `lruConnections`: `LinkedHashMap` 类型, 用来记录各个连接的使用情况, 并据此关闭空闲时间超过 `connectionsMaxIdleNanos` 的连接。

下面介绍 `KSelector` 的核心方法。`KSelector.connect()` 方法主要负责创建 `KafkaChannel`, 并添加到 `channels` 集合中保存。其代码如下:

```
public void connect(...) throws IOException {
    SocketChannel socketChannel = SocketChannel.open(); // 创建 SocketChannel
    socketChannel.configureBlocking(false); // 配置成非阻塞模式
    Socket socket = socketChannel.socket();
    socket.setKeepAlive(true); // 设置为长连接
    socket.setSendBufferSize(sendBufferSize); // 设置 SO_SNDBUF 大小
    socket.setReceiveBufferSize(receiveBufferSize); // 设置 SO_RCVBUF 大小
    // 因为是非阻塞方式, 所以 SocketChannel.connect() 方法是发起一个连接, connect 方
    // 法在连接正式建立之前就可能返回, 在后面会通过 KSelector.finishConnect() 方法确认连
    // 接是否真正建立了
    connected = socketChannel.connect(address);
    // 异常处理 (略)
    // 将这个 socketChannel 注册到 nioSelector 上, 并关注 OP_CONNECT 事件
    SelectionKey key = socketChannel.register(nioSelector, SelectionKey.OP_
CONNECT);
    // 创建 KafkaChannel
```



```

    KafkaChannel channel = channelBuilder.buildChannel(id, key,
maxReceiveSize);
    key.attach(channel); // 将 KafkaChannel 注册到 key 上
    // 将 NodeId 和 KafkaChannel 绑定, 放到 channels 中管理
    this.channels.put(id, channel);
}

```

KSelector.send() 方法是将之前创建的 RequestSend 对象缓存到 KafkaChannel 的 send 字段中, 并开始关注此连接的 OP_WRITE 事件, 并没有发生网络 I/O。在下次调用 KSelector.poll() 时, 才会将 RequestSend 对象发送出去。如果此 KafkaChannel 的 send 字段上还保存着一个未完全发送成功的 RequestSend 请求, 为防止覆盖数据, 则会抛出异常。也就是说, 每个 KafkaChannel 一次 poll 过程中只能发送一个 Send 请求。

KSelector.poll() 方法真正执行网络 I/O 的地方, 它会调用 nioSelector.select() 方法等待 I/O 事件发生。当 Channel 可写时, 发送 KafkaChannel.send 字段 (切记, 一次最多只发送一个 RequestSend, 有时候一个 RequestSend 也发送不完, 需要多次 poll 才能发送完成); Channel 可读时, 读取数据到 KafkaChannel.receive, 读取一个完整的 NetworkReceive 后, 会将其缓存到 stagedReceives 中, 当一次 pollSelectionKeys() 完成后会将 stagedReceives 中的数据转移到 completedReceives。最后调用 maybeCloseOldestConnection() 方法, 根据 lruConnections 记录和 connectionsMaxIdleNanos 最大空闲时间, 关闭长期空闲的连接。下面是 KSelector.poll() 方法的代码:

```

public void poll(long timeout) throws IOException {
    clear(); // 将上一次 poll() 方法的结果全部清除掉
    // 调用 nioSelector.select() 方法, 等待 I/O 事件发生
    int readyKeys = select(timeout);
    // 省略一些统计操作
    if (readyKeys > 0 || !immediatelyConnectedKeys.isEmpty()) {
        // 处理 I/O 事件
        pollSelectionKeys(this.nioSelector.selectedKeys(), false);
        pollSelectionKeys(immediatelyConnectedKeys, true);
    }
    addToCompletedReceives(); // 将 stagedReceives 复制到 completedReceives 集合中
    maybeCloseOldestConnection(); // 关闭长期空闲的连接
}

```

KSelector.pollSelectionKeys() 方法是处理 I/O 操作的核心方法, 其中会分别处理 OP_

CONNECT、OP_READ、OP_WRITE 事件，并且会检测连接状态。下面是其代码：

```
private void pollSelectionKeys(Iterable<SelectionKey> selectionKeys,
    boolean isImmediatelyConnected) {

    Iterator<SelectionKey> iterator = selectionKeys.iterator();
    while (iterator.hasNext()) {
        SelectionKey key = iterator.next();
        iterator.remove();
        // 之前创建连接时，将 KafkaChannel 注册到 key 上，就是为了在这里获取
        KafkaChannel channel = channel(key);
        lruConnections.put(channel.id(), currentTimeNanos); // 更新 lru 信息
        try {
            // 对 connect 方法返回 true 或 OP_CONNECTION 事件的处理
            if (isImmediatelyConnected || key.isConnectable()) {
                // finishConnect 方法会先检测 sockChannel 是否建立完成，建立后，会取消对
                // OP_CONNECT 事件关注，开始关注 OP_READ 事件
                if (channel.finishConnect()) {
                    this.connected.add(channel.id()); // 添加到“已连接”的集合中
                } else
                    continue; // 连接未完成，则跳过对此 Channel 的后续处理
            }
            // 调用 KafkaChannel.prepare() 方法进行身份验证，在第 4 章会详细介绍身份验证的内容
            if (channel.isConnected() && !channel.ready())
                channel.prepare();

            if (channel.ready() && key.isReadable() && !hasStagedReceive(channel)) {
                // OP_READ 事件处理
                NetworkReceive networkReceive;
                while ((networkReceive = channel.read()) != null) {
                    // 上面 channel.read() 读取到一个完整的 NetworkReceive，则将其添加
                    // 到 stagedReceives 中保存
                    // 若读取不到一个完整的 NetworkReceive，则返回 null，下次处理 OP_READ 事件时，继续
                    // 读取，直至读取到一个完整的 NetworkReceive
                    addToStagedReceives(channel, networkReceive);
                }
            }
        }
    }
}
```

```

        if (channel.ready() && key.isWritable()) { // OP_WRITE 事件处理
            Send send = channel.write();
            // 上面的 channel.write() 方法将 KafkaChannel.send 字段发送出去, 如果
            // 未完成发送, 则返回 null, 如果发送完成, 则返回 send, 并添加到
            // completeSends 集合中, 待后续处理
            if (send != null) {
                this.completedSends.add(send); // 添加到 completedSends 集合
            }
        }
        // completedSends 和 completedReceives 分别表示在 Selector 端已经发送的
        // 和接收到的请求, 它们会在 NetworkClient 的 poll 调用之后被不同的
        // handleCompleteXXX() 方法处理
    } catch (Exception e) {
        // 抛出异常, 则认为连接关闭, 将对应 NodeId 添加到 disconnected 集合
        close(channel);
        this.disconnected.add(channel.id());
    }
}
}

```

最终, 读写操作还是交给了 `KafkaChannel`, 下面来分析其相关的方法:

```

public class KafkaChannel {
    ....
    public void setSend(Send send) {
        if (this.send != null) throw new IllegalStateException("...");
        this.send = send; // 设置 send 字段
        // 关注 OP_WRITE 事件
        this.transportLayer.addInterestOps(SelectionKey.OP_WRITE);
    }

    private boolean send(Send send) throws IOException {
        // 如果 send 在一次 write 调用时没有发送完, SelectionKey 的 OP_WRITE 事件没有取
        // 消, 还会继续监听此 Channel 的 OP_WRITE 事件, 直到整个 send 请求发送完毕才取消
        send.writeTo(transportLayer);
        // 判断发送是否完成是通过 ByteBuffer 中是否还有剩余字节来判断的
        if (send.completed())

```



```

        transportLayer.removeInterestOps(SelectionKey.OP_WRITE);

        return send.completed();
    }

    public NetworkReceive read() throws IOException {
        NetworkReceive result = null;
        // 初始化 NetworkReceive (略)
        // receive() 方法从 transportLayer 中读取数据到 NetworkReceive 对象中。假设
        // 并没有读完一个完整的 NetworkReceive, 则下次触发 OP_READ 事件时继续填充此
        // NetworkReceive 对象; 如果读取了一个完整的 NetworkReceive 对象, 则将
        // receive 置空, 下次触发读操作时, 创建新 NetworkReceive 对象
        receive(receive);
        if (receive.complete()) {
            receive.payload().rewind();
            result = receive;
            receive = null;
        }
        return result;
    }
}

```

Send 和 NetworkReceive 这两个类是对 ByteBuffer 的封装, 比较简单, 留给读者自己分析。需要注意的是, 在 NetworkReceive 从连接读取数据的时候, 是先读取消息的头部, 其中封装了消息长度, 再按照其长度创建合适大小的 ByteBuffer, 然后读取消息体。

KSelector.pollSelectionKeys() 方法通过 selectionKey.isValid() 的返回值以及执行过程中是否抛出异常来判断连接的状态, 并将断开的连接收集到 disconnected 集合, 并在后续操作中进行重连。

2.4.3 InFlightRequests

InFlightRequests 队列的主要作用是缓存了已经发出去但没收到响应的 ClientRequest。其底层是通过一个 Map<String, Deque<ClientRequest>> 对象实现的, key 是 NodeId, value 是发送到对应 Node 的 ClientRequest 对象集合。InFlightRequests 提供了很多管理这个缓存队列的方法, 还通过配置参数, 限制了每个连接最多缓存的 ClientRequest 个数。InFlightRequests 的结构如图 2-25 所示。

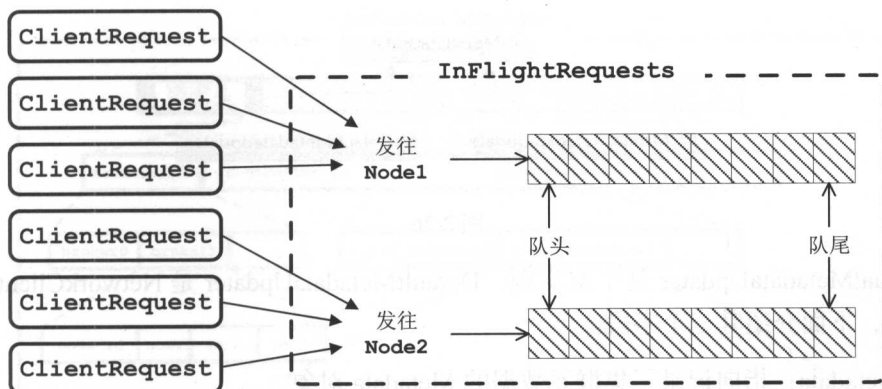


图 2-25

`InFlightRequests.canSendMore()` 方法比较重要, `NetworkClient` 调用此方法是用于判断是否可以向指定 Node 发送请求的条件之一, 其代码如下:

```
public boolean canSendMore (String node) {
    Deque<ClientRequest> queue = requests.get(node);
    return queue == null
        || queue.isEmpty()
        || (queue.peekFirst().request().completed()
            && queue.size() < this.maxInFlightRequestsPerConnection);
}
```

`queue == null` 和 `queue.isEmpty()` 这两个条件比较容易理解, 不再赘述, `queue.peekFirst().request().completed()` 这个条件为 `true` 表示当前队头的请求已经发送完成, 如果队头的请求迟迟发送不出去, 可能是网络出现问题, 则不能继续向此 Node 发送请求。此外, 队头的消息与对应 `KafkaChannel.send` 字段指向的是同一个消息, 为了避免未发送的消息被覆盖, 也不能让 `KafkaChannel.send` 字段指向新请求。最后 `queue.size() < this.maxInFlightRequestsPerConnection` 条件则是为了判断 `InFlightRequests` 队列中是否堆积过多请求。如果 Node 已经堆积了很多未响应的请求, 说明这个节点负载可能较大或是网络连接有问题, 继续向其发送请求, 则可能导致请求超时。

2.4.4 MetadataUpdater

`MetadataUpdater` 接口是一个辅助 `NetworkClient` 更新的 `Metadata` 的接口, 它有两个实现类, 如图 2-26 所示。

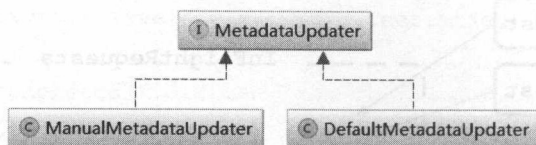


图 2-26

ManualMetadataUpdater 是个空实现，DefaultMetadataUpdater 是 NetworkClient 使用的默认实现，下面介绍其三个字段。

- metadata: 指向记录了集群元数据的 Metadata 对象。
- metadataFetchInProgress: 用来标识是否已经发送了 MetadataRequest 请求更新 Metadata，如果已经发送，则没必要重复发送。
- lastNoNodeAvailableMs: 当检测到没有可用节点时，会用此字段记录时间戳。

maybeUpdate() 方法是 DefaultMetadataUpdater 的核心方法，用来判断当前的 Metadata 中保存的集群元数据是否需要更新。首先检测 metadataFetchInProgress 字段，如果没发送，满足下面任一条件即可更新：

- Metadata.needUpdate 字段被设置为 true，且退避时间已到。
- 长时间没更新，默认 5 分钟更新一次。

如果需要更新，则发送 MetadataRequest 请求，MetadataRequest 请求的格式比较简单，其消息头部包含 ApiKeys.METADATA 标识，消息体中包含 Topic 集合表示需要获取元数据的 Topic，如果 Topic 集合为 null 则表示请求全部 Topic 的元数据。MetadataResponse 的格式略显复杂，如图 2-27 所示。

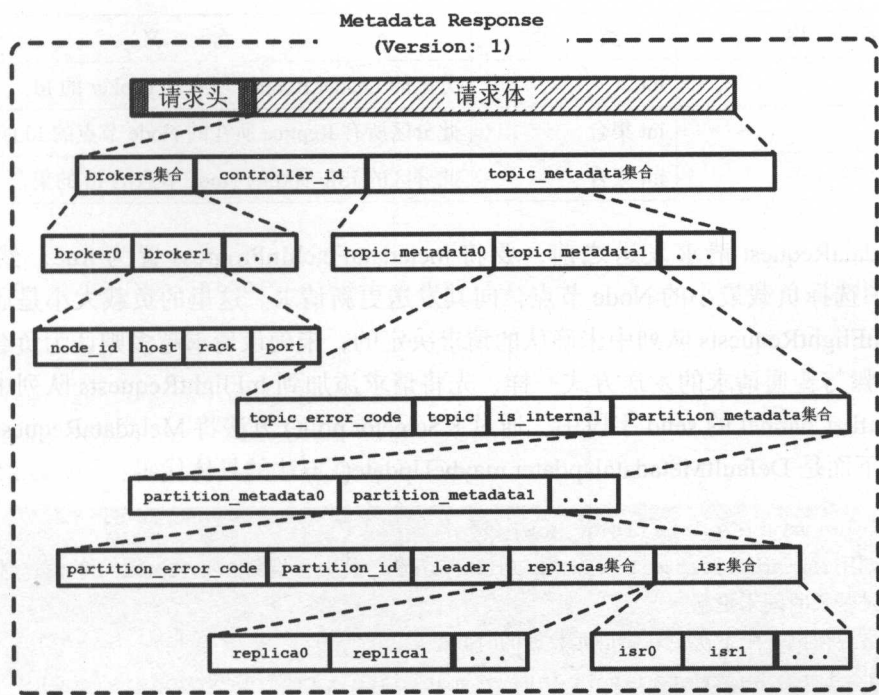


图 2-27

Metadata Response(Version: 1) 请求体各个字段的含义如表 2-3 所示。

表 2-3

名 称	类 型	含 义
node_id	int	Node 节点的 Id
host	String	Node 节点的 Host 名称
port	int	Node 节点的端口号
rack	String	每个 Broker 的机架信息（可为 null）
controller_id	int	controller 所在的 Node 节点的 Id
topic_error_code	short	错误码
topic	String	Topic 的名称
is_internal	boolean	是否为 Kafka 内部的 Topic
partition_error_code	short	错误码
partition_id	int	分区编号

名 称	类 型	含 义
leader	int	分区的 Leader Replica 所在的 Broker 的 Id
replicas	int 集合	此分区所有 Replica 所在的 Node 节点的 Id 的集合
isr	int 集合	此分区的 ISR 所在的 Node 节点的 Id 的集合

MetadataRequest 请求发送之前, 要将 metadataFetchInProgress 置为 true, 然后从所有 Node 中选择负载最小的 Node 节点, 向其发送更新请求。这里的负载大小是通过每个 Node 在 InFlightRequests 队列中未确认的请求决定的, 未确认请求越多则认为负载越大。剩余的步骤与普通请求的发送方式一样, 先将请求添加到 InFlightRequests 队列中, 然后设置到 KafkaChannel 的 send 字段中, 通过 KSelector.poll() 方法将 MetadataRequest 请求发送出去。下面是 DefaultMetadataUpdater.maybeUpdate() 方法的具体代码:

```
public long maybeUpdate(long now) {
    // 调用 Metadata.timeToNextUpdate() 方法, 其中会检测 needUpdate 的值、退避时间、
    // 是否长时间未更新
    // 最终得到一个下次更新集群元数据的时间戳
    long timeToNextMetadataUpdate = metadata.timeToNextUpdate(now);

    // 获取下次尝试重新连接服务端的时间戳
    long timeToNextReconnectAttempt = Math.max(this.lastNoNodeAvailableMs
        + metadata.refreshBackoff() - now, 0);

    // 检测是否已经发送了 MetadataRequest 请求
    long waitForMetadataFetch = this.metadataFetchInProgress ? Integer.
MAX_VALUE : 0;

    // 计算当前距离下次可以发送 MetadataRequest 请求的时间差
    long metadataTimeout = Math.max(Math.max(timeToNextMetadataUpdate,
        timeToNextReconnectAttempt), waitForMetadataFetch);
    if (metadataTimeout == 0) { // 允许发送 MetadataRequest 请求
        // 找到负载最小的 Node, 若没有可用 Node, 则返回 null
        Node node = leastLoadedNode(now);
        // 创建并缓存 MetadataRequest, 等待下次 poll() 方法才会真正发送
        maybeUpdate(now, node);
    }

    return metadataTimeout;
}
```

```

}

private void maybeUpdate(long now, Node node) {
    if (node == null) { // 检测是否有 Node 可用
        this.lastNoNodeAvailableMs = now; // 设置 lastNoNodeAvailableMs
        return;
    }
    String nodeConnectionId = node.idString();

    // 检测是否允许向此 Node 发送请求, 下一节详细介绍
    if (canSendRequest(nodeConnectionId)) {
        this.metadataFetchInProgress = true;
        MetadataRequest metadataRequest;
        if (metadata.needMetadataForAllTopics()) // 指定需要更新元数据的 Topic
            metadataRequest = MetadataRequest.allTopics();
        else
            metadataRequest = new MetadataRequest(new ArrayList<>(metadata.
topics()));
        // 将 MetadataRequest 封装成 ClientRequest...
        ClientRequest clientRequest = request(now, nodeConnectionId,
            metadataRequest);

        doSend(clientRequest, now); // 缓存请求, 在下次 poll() 操作中会将其发送出去
    } else if (connectionStates.canConnect(nodeConnectionId, now)) {
        initiateConnect(node, now); // 初始化连接
    } else { // 已成功连接到指定节点, 但不能发送请求, 则更新 lastNoNodeAvailableMs 后等待
        this.lastNoNodeAvailableMs = now;
    }
}
}

```

在收到 `MetadataResponse` 之后, 会先调用 `MetaUpdater.maybeHandleCompletedReceive()` 方法检测是否为 `MetadataResponse`, 如果是, 则调用 `handleResponse()` 解析响应, 并构造 `Cluster` 对象更新 `Metadata.cluster` 字段。注意, `Cluster` 是不可变对象, 所以更新集群元数据的方式是: 创建新的 `Cluster` 对象, 并覆盖 `Metadata.cluster` 字段。具体代码如下:

```

public boolean maybeHandleCompletedReceive(ClientRequest req, long now,
    Struct body) {
    short apiKey = req.request().header().apiKey();
    // 检测是否为 MetadataRequest 请求
    if (apiKey == ApiKeys.METADATA.id && req.isInitiatedByNetworkClient()) {
        handleResponse(req.request().header(), body, now);
        return true;
    }
    return false;
}

private void handleResponse(RequestHeader header, Struct body, long now) {
    this.metadataFetchInProgress = false; // 修改 metadataFetchInProgress
    // 解析 MetadataResponse
    MetadataResponse response = new MetadataResponse(body);
    Cluster cluster = response.cluster(); // 创建 Cluster 对象
    ... .. // 检测 MetadataResponse 中的错误码 (略)

    if (cluster.nodes().size() > 0) {
        // 在 Metadata.update() 方法中, 首先通知 Metadata 上的监听器, 之后更新 cluster
        // 字段, 最后唤醒等待 Metadata 更新完成的线程
        this.metadata.update(cluster, now);
    } else {
        // 更新 metadata 失败, 只是更新 lastRefreshMs 字段
        this.metadata.failedUpdate(now);
    }
}

```

当连接断开或其他异常导致无法获取到响应时, 由 `maybeHandleDisconnection()` 方法处理, 它会将 `metadataFetchInProgress` 字段置为 `false`, 这样就可以顺利发送下一次更新 Metadata 请求了。


```

public boolean maybeHandleDisconnection(ClientRequest request) {
    ApiKeys requestKey = ApiKeys.forId(request.request().header().
apiKey());
    if (requestKey == ApiKeys.METADATA) { // 检测是否为 MetadataRequest 请求
        ... .. // 输出日志
        metadataFetchInProgress = false; // 更新 metadataFetchInProgress
        return true;
    }
    return false;
}

```

2.4.5 NetworkClient

NetworkClient 中所有连接的状态由 ClusterConnectionStates 管理，它底层使用 Map<String, NodeConnectionState> 实现，key 是 NodeId，value 是 NodeConnectionState 对象，其中使用 ConnectionState 枚举表示连接状态，还记录了最近一次尝试连接的时间戳。

前面已经介绍完了 NetworkClient 依赖的组件，下面来看一下 NetworkClient 的实现。NetworkClient 是一个通用的网络客户端实现，不只用于生产者发送消息，也可以用于消费者消费消息以及服务端 Broker 之间的通信。

下面介绍 NetworkClient 的核心方法。NetworkClient.ready() 方法用来检查 Node 是否准备好接收数据。首先通过 NetworkClient.isReady() 方法检查是否可以向一个 Node 发送请求，需要符合以下三个条件，则表示 Node 已准备好：

- Metadata 并未处于正在更新或需要更新的状态。
- 已经成功建立连接且连接正常 connectionStates.isConnected(node)。
- InFlightRequests.canSendMore() 返回 true。

如果 NetworkClient.isReady() 返回 false，且满足下面两个条件，则会调用 initiateConnect() 方法发起连接。

- 连接不能是 CONNECTING 状态，必须是 DISCONNECTED。
- 为了避免网络拥塞，重连不能太频繁，两次重试之间的时间差必须大于重试的退避时间，由 reconnectBackoffMs 字段指定。

NetworkClient.initiateConnect() 方法会修改在 ClusterConnectionStates 中的连接状态，

并调用 `Selector.connect()` 方法发起连接。之后调用 `Selector.pollSelectionKeys()` 方法时，判断连接是否建立。如果建立成功，则会将 `ConnectionState` 设置为 `CONNECTED`。

`NetworkClient.send()` 方法主要是将请求设置到 `KafkaChannel.send` 字段，同时将请求添加到 `InFlightRequests` 队列中等待响应。

```
public void send(ClientRequest request, long now) {
    String nodeId = request.request().destination();
    if (!canSendRequest(nodeId)) // 检测是否能够向指定 Node 发送请求
        throw new IllegalStateException("...");

    // 设置 KafkaChannel.send 字段，并将请求放入 InFlightRequests 等待响应
    doSend(request, now);
}

// canSendRequest() 方法的代码如下：
private boolean canSendRequest(String node) {
    return connectionStates.isConnected(node) // 检测连接状态
        && selector.isChannelReady(node) // 检测网络协议正常且是否通过了身份认证
        && inFlightRequests.canSendMore(node); // 参考 InFlightRequests 小节
}

// doSend() 方法的代码如下：
private void doSend(ClientRequest request, long now) {
    request.setSendTimeMs(now);
    // 将请求添加到 inFlightRequests 队列中等待响应
    this.inFlightRequests.add(request);
    selector.send(request.request()); // 参考 KSelector 小节
}
```

`NetworkClient.poll()` 方法调用 `KSelector.poll()` 进行网络 I/O（参考 `KSelector` 小节），并使用 `handle*()` 方法对 `KSelector.poll()` 产生的各种数据和队列进行处理。

```

public List<ClientResponse> poll(long timeout, long now) {
    long metadataTimeout = metadataUpdater.maybeUpdate(now); // 更新Metadata
    this.selector.poll(...); // 执行 I/O 操作
    List<ClientResponse> responses = new ArrayList<>(); // 响应队列
    handleCompletedSends(responses, updatedNow); // 处理 completedSends 队列
    handleCompletedReceives(responses, updatedNow); // 处理 completedReceives 队列
    handleDisconnections(responses, updatedNow); // 处理 disconnected 列表
    handleConnections(); // 处理 connected 列表
    // 处理 InFlightRequests 中的超时请求
    handleTimedOutRequests(responses, updatedNow);
    // 循环调用 ClientRequest 的回调函数
    for (ClientResponse response : responses) {
        if (response.request().hasCallback()) {
            try {
                // 这个 callback 会调用 Sender.handleProduceResponse() 方法
                response.request().callback().onComplete(response);
            } catch (Exception e) {log.error("...", e); }
        }
    }
    return responses;
}

```

下面来看一下 handle*() 方法的处理逻辑：

- handleCompletedSends() 方法：首先，InFlightRequests 保存的是已发送但没收到响应的请求，completedSends 保存的是最近一次 poll() 方法中发送成功的请求，所以 completedSends 列表与 InFlightRequests 中对应队列的最后一个请求应该是一致的，如图 2-28 所示。

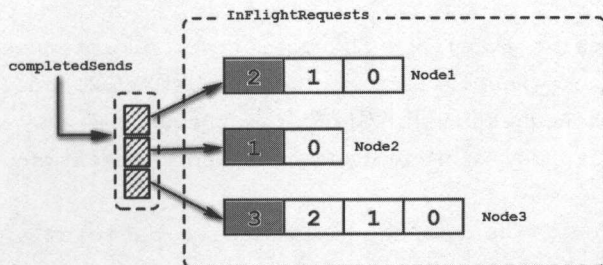


图 2-28

`handleCompletedSends()` 方法会遍历 `completedSends`，如果发现不需要响应的请求，则将其从 `InFlightRequests` 中删除，并向 `responses` 列表中添加对应的 `ClientResponse`，在 `ClientResponse` 中包含一个指向 `ClientRequest` 的引用。`handleCompletedSends()` 方法的代码如下：

```
private void handleCompletedSends(List<ClientResponse> responses, long now) {

    // 遍历 completedSends 集合
    for (Send send : this.selector.completedSends()) {
        ClientRequest request = this.inFlightRequests.lastSent(
            send.destination()); // 获取指定队列的第一个元素

        if (!request.expectResponse()) { // 检测请求是否需要响应
            // 将 inFlightRequests 中对应队列中的第一个请求删除
            this.inFlightRequests.completeLastSent(send.destination());
            // 生成 ClientResponse 对象，添加到 responses 集合
            responses.add(new ClientResponse(request, now, false, null));
        }
    }
}
```

- `handleCompletedReceives()` 方法：遍历 `completedReceives` 队列，并在 `InFlightRequests` 中删除对应的 `ClientRequest`，并向 `responses` 列表中添加对应的 `ClientResponse`。如果是 `Metadata` 更新请求的响应，则会调用 `MetadataUpdater` 中的 `maybeHandleCompletedReceive()` 方法，更新 `Metadata` 中记录的集 Kafka 集群元数据。

```
private void handleCompletedReceives(List<ClientResponse> responses, long now) {
    for (NetworkReceive receive : this.selector.completedReceives()) {
        String source = receive.source(); // 获取返回响应的 NodeId
        // 从 InFlightRequests 中取出对应的 ClientRequest
        ClientRequest req = inFlightRequests.completeNext(source);
        // 解析响应
        Struct body = parseResponse(receive.payload(), req.request().header());
        // 调用 MetadataUpdater.maybeHandleCompletedReceive() 方法处理
    }
}
```



```
// MetadataResponse, 其中会更新 Metadata 中记录的集群元数据, 并唤醒所有等待 Metadata
// 更新完成的线程
if (!metadataUpdater.maybeHandleCompletedReceive(req, now, body)) {
    // 如果不是 MetadataResponse, 则创建 ClientResponse 并添加到 responses 集合
    responses.add(new ClientResponse(req, now, false, body));
}
}
}
```

- `handleDisconnections()` 方法: 遍历 `disconnected` 列表, 将 `InFlightRequests` 对应节点的 `ClientRequest` 清空, 对每个请求都创建 `ClientResponse` 并添加到 `responses` 列表中。这里创建的 `ClientResponse` 会标识此响应并不是服务端返回的正常响应, 而是因为连接断开产生的。如果是 `Metadata` 更新请求的响应, 则会调用 `MetadataUpdater` 中的 `maybeHandleDisconnection()` 方法处理。最后将 `Metadata.needUpdate` 设置为 `true`, 标识需要更新集群元数据。

```
private void handleDisconnections(List<ClientResponse> responses, long
now) {
    // 更新连接状态, 并清理掉 InFlightRequests 中断开连接的 Node 对应的 ClientRequest
    for (String node : this.selector.disconnected()) {
        processDisconnection(responses, node, now);
    }

    if (this.selector.disconnected().size() > 0)
        metadataUpdater.requestUpdate(); // 标识需要更新集群元数据
}

private void processDisconnection(List<ClientResponse> responses,
String nodeId, long now) {
    connectionStates.disconnected(nodeId, now); // 更新连接状态
    for (ClientRequest request : this.inFlightRequests.clearAll(nodeId)) {
        // 调用 MetadataUpdater.maybeHandleDisconnection() 方法处理 MetadataRequest
        if (!metadataUpdater.maybeHandleDisconnection(request))
            // 如果不是 MetadataRequest, 则创建 ClientResponse 并添加到 responses 集合
            // 注意第三个参数, 表示连接是否断开
            responses.add(new ClientResponse(request, now, true, null));
    }
}
```

- `handleConnections()` 方法：遍历 `connected` 列表，将 `ConnectionStates` 中记录的连接状态修改为 `CONNECTED`。代码比较简单，不再贴出来了。
- `handleTimedOutRequests()` 方法：遍历 `InFlightRequests` 集合，获取有超时请求的 `Node` 集合，之后的处理逻辑与 `handleDisconnections()` 方法一样，代码就不贴出来了。

经过一系列 `handle*()` 方法处理后，`NetworkClient.poll()` 方法中产生的全部 `ClientResponse` 已经被收集到 `responses` 列表中。之后，遍历 `responses` 调用每个 `ClientRequest` 中记录的回调，如果是异常响应则请求重发，如果是正常响应则调用每个消息的自定义 `Callback`。在前面的 `createProduceRequests()` 方法中提到过，这里调用的 `Callback` 回调对象，也就是 `RequestCompletionHandler` 对象，其 `onComplete()` 方法最终调用 `Sender.handleProduceResponse()` 方法，其逻辑如下：

(1) 如果因为断开连接或异常而产生的响应：

a) 遍历 `ClientRequest` 中的 `RecordBatch`，则尝试将 `RecordBatch` 重新加入 `RecordAccumulator`，重新发送。

b) 如果异常类型不允许重试或重试次数达到上限，则执行 `RecordBatch.done()` 方法，此方法会循环调用 `RecordBatch` 中每个消息的 `Callback` 函数，并将 `RecordBatch` 的 `produceFuture` 设置为“异常完成”。最后，释放 `RecordBatch` 底层的 `ByteBuffer`。

c) 最后，根据异常类型，决定是否设置更新 `Metadata` 标志。

(2) 如果是服务端正常的响应或不需要响应的情况：

a) 解析响应。

b) 遍历对应 `ClientRequest` 中的 `RecordBatch`，执行 `RecordBatch.done()` 方法。

c) 释放 `RecordBatch` 底层的 `ByteBuffer`。

下面是 `Sender.handleProduceResponse()` 方法的具体代码：

```
private void handleProduceResponse(ClientResponse response,
                                   Map<TopicPartition, RecordBatch>
batches, long now) {
    int correlationId = response.request().request().header().
correlationId();
    // 对于连接断开而产生的 ClientResponse，会重试发送请求，若不能重试，则调用其中每
    // 条消息的回调
    if (response.wasDisconnected()) {
```



```

    for (RecordBatch batch : batches.values())
        completeBatch(batch, Errors.NETWORK_EXCEPTION, -1L,
            Record.NO_TIMESTAMP, correlationId, now);
} else {
    if (response.hasResponse()) {
        ProduceResponse produceResponse = new ProduceResponse(
            response.responseBody());
        for (Map.Entry<TopicPartition, ProduceResponse.
PartitionResponse> entry :
            produceResponse.responses().entrySet()) {
            TopicPartition tp = entry.getKey();
            ProduceResponse.PartitionResponse partResp = entry.
getValue();

            Errors error = Errors.forCode(partResp.errorCode);
            RecordBatch batch = batches.get(tp);

            // 调用 completeBatch() 方法处理
            completeBatch(batch, error, partResp.baseOffset,
                partResp.timestamp, correlationId, now);
        }
    } else { // 不需要响应的请求, 直接调用 completeBatch() 方法处理
        for (RecordBatch batch : batches.values())
            completeBatch(batch, Errors.NONE, -1L, Record.NO_
TIMESTAMP,
                correlationId, now);
    }
}

// 下面是 completeBatch() 方法的实现
private void completeBatch(RecordBatch batch, Errors error, long
baseOffset,
long timestamp, long correlationId, long now) {

    if (error != Errors.NONE && canRetry(batch, error)) {
        // 对于可重试的 RecordBatch, 则重新添加到 RecordAccumulator 中, 等待发送
        this.accumulator.reenqueue(batch, now);
    } else {
        // 不能重试的话, 会将 RecordBatch 都标记为“异常完成”, 并释放 RecordBatch

```

```

        RuntimeException exception;

        if (error == Errors.TOPIC_AUTHORIZATION_FAILED) // 获取异常
            exception = new TopicAuthorizationException(batch.topicPartition.topic());
        else
            exception = error.exception();
        // 调用 RecordBatch.done() 方法, 调用消息的回调函数
        batch.done(baseOffset, timestamp, exception);

        this.accumulator.deallocate(batch); // 释放空间
    }
    if (error.exception() instanceof InvalidMetadataException)
        metadata.requestUpdate(); // 标识需要更新 Metadata 中记录的集群元数据
}

```

本章小结

首先, 本章通过一个生产者示例程序向读者介绍了如何通过 `KafkaProducer` 的 API 实现 Kafka 生产者。然后, 详细地分析了整个 `KafkaProducer` 的架构特点和关键细节, 剖析了 `KafkaProducer` 拦截消息、序列化消息、路由消息功能的实现原理, 介绍了 `RecordAccumulator` 的结构和实现以及它如何缓存消息。最后从如何 `RecordAccumulator` 中批量获取消息, 如何组织请求, 如何利用 `NetworkClient` 管理与服务端的连接, 如何发送请求并处理正常响应和异常响应, 如何更新 Kafka 集群元数据等几个方面介绍了 `Sender` 线程。通过本章的分析, 希望读者可以理解生产者的设计原理和实现细节, 在实践中更好地应用 `KafkaProducer`。

第 3 章

消费者

3.1 KafkaConsumer 使用示例

Kafka 不仅提供了用于实现生产者的 `KafkaProducer`，还提供了用于实现消费者的 `KafkaConsumer`。消费者的根本目的是从 Kafka 服务端拉取消息，并交给业务逻辑进行处理。`KafkaConsumer` 提供了一套封装良好的 API，开发人员可以基于这套 API 轻松实现从 Kafka 服务端拉取消息的功能，这样开发人员不必关心与 Kafka 服务端之间网络连接的管理、心跳检测、请求超时重试等底层操作，也不必关心订阅 Topic 的分区数量、分区 Leader 副本的网络拓扑以及 Consumer Group 的 Rebalance 等 Kafka 的具体细节，`KafkaConsumer` 中还提供了自动提交 offset 的功能，当开发人员从这些烦琐的细节中解脱出来之后，就可以更加关注业务逻辑，提高了开发效率。

在 Kafka core 模块的 `kafka.consumer` 包中，还保留着旧版本的消费者实现，它由 Scala 实现，此客户端已经不再推荐大家使用了，在后续版本中会将其标记为 “deprecated”，在主版本升级后可能会将其删除。新版本的消费者客户端实现 `KafkaConsumer`（Java 实现）是在 Kafka clients 模块的 `org.apache.kafka.clients.consumer` 包中。本章将以新版本的 `KafkaConsumer` 为基础展开介绍，学习 `KafkaConsumer` 的使用和相关配置，并深入剖析其核心代码及原理。对于旧版本的消费者客户端本书不做详细介绍。

下面通过一个示例程序介绍 `KafkaConsumer` 的使用：

```

public class KafkaConsumerDemo{
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092"); // Broker 的地址
        props.put("group.id", "test"); // 所属 Consumer Group 的 Id
        props.put("enable.auto.commit", "true"); // 自动提交 offset
        // 自动提交 offset 的时间间隔
        props.put("auto.commit.interval.ms", "1000");
        props.put("session.timeout.ms", "30000");
        // key 使用的 Deserializer, 参考第 2 章的相关小节
        props.put("key.deserializer",
            "org.apache.kafka.common.serialization.StringDeserializer");
        // value 使用的 Deserializer, 参考第 2 章的相关小节
        props.put("value.deserializer",
            "org.apache.kafka.common.serialization.StringDeserializer");
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        // 订阅 test1 和 test2 两个 Topic
        consumer.subscribe(Arrays.asList("test1", "test2"));
        try {
            while (true) {
                // 从服务端拉取消息, 每次 poll() 可以拉取多个消息
                ConsumerRecords<String, String> records = consumer.poll(100);
                // 消费消息, 这里仅仅是将消息的 offset、key、value 输出
                for (ConsumerRecord<String, String> record : records)
                    System.out.printf("offset = %d, key = %s, value = %s\n",
                        record.offset(), record.key(), record.value());
            }
        } finally {
            consumer.close(); // 关闭 Consumer
        }
    }
}

```

通过上面的程序, 我们也可以看出 `KafkaConsumer` 的核心方法是 `poll()` 方法, 它负责从 Kafka 服务端拉取消息。

3.2 传递保证语义 (Delivery guarantee semantic)

在第1章介绍 Kafka 核心概念时提到过, Kafka 服务端并不会记录消费者的消费位置, 而是由消费者自己决定如何保存如何记录其消费的 offset。旧版本的消费者会将其消费位置记录到 ZooKeeper 中, 在新版本消费者中为了缓解 ZooKeeper 集群的压力, 在 Kafka 服务端中添加了一个名为 “__consumer_offsets” 的内部 Topic, 为了便于描述, 本文借鉴源码注释中的简称 “Offsets Topic”。Offsets Topic 可以用来保存消费者提交的 offset, 当出现消费者上/下线时会触发 Consumer Group 进行 Rebalance 操作, 对分区进行重新分配, 待 Rebalance 操作完成后, 消费者就可以读取 Offsets Topic 中记录的 offset, 并从此 offset 位置继续消费。当然, 使用 Offsets Topic 记录消费者的 offset 只是默认选项, 开发人员可以根据业务需求将 offset 记录在别的存储中。

在消费者消费消息的过程中, 提交 offset 的时机显得非常重要, 因为它决定了消费者故障重启后的消费位置。在上一节的示例程序中, 我们通过将 `enable.auto.commit` 选项设置为 `true` 可以起到自动提交 offset 的功能, `auto.commit.interval.ms` 选项则设置了自动提交的时间间隔, 这是最简单的提交 offset 方式。每次在调用 `KafkaConsumer.poll()` 方法时都会检测是否需要自动提交, 并提交上次 `poll()` 方法返回的最后一个消息的 offset。为了避免消息丢失, 建议 `poll()` 方法之前要处理完上次 `poll` 方法拉取的全部消息。KafkaConsumer 中还提供了两个手动提交 offset 的方法, 分别是 `commitSync()` 方法和 `commitAsync()` 方法, 它们都可以指定提交的 offset 值, 区别在于前者是同步提交, 后者是异步提交。提交 offset 的具体原理和实现稍后分析。

下面来介绍消息的传递保证 (Delivery guarantee semantic) 的相关内容, 传递保证语义有以下三个级别。

- At most once: 消息可能会丢, 但绝不会重复传递。
- At least once: 消息绝不会丢, 但可能会重复传递。
- Exactly once: 每条消息只会被传递一次。

在实践中很少出现对 “At most once” 的需求, 而在很多场景中, “Exactly once” 语义才是我们需要的, 所以我们详述这种语义。当然, 如果通过 Kafka 传递的消息是幂等性的 (即一条消息被反复消费多次并不会对计算结果产生影响), 使用 “At least once” 语义也是没有问题的。“Exactly once” 语义由生产者和消费者两部分共同决定: 首先, 生产者要保证不会产生重复的消息; 其次, 消费者不能重复拉取相同的消息。

先来讨论生产者部分, 当生产者向 Kafka 发送消息, 且正常得到响应的时候, 可以确

保生产者不会产生重复的消息。但是，如果生产者发送消息后，遇到网络问题，无法获取响应，生产者就无法判断该消息是否成功提交给了 Kafka。在第 2 章分析生产者的机制时，我们知道，当出现异常时，会进行消息重传，这就可能出现“*At least one*”语义。为了实现“*Exactly once*”语义，这里提供两个可选方案：

- 每个分区只有一个生产者写入消息，当出现异常或超时的情况时，生产者就要查询此分区的最后一个消息，用来决定后续操作是消息重传还是继续发送。
- 为每个消息添加一个全局唯一主键，生产者不做其他特殊处理，按照之前分析方式进行重传，由消费者对消息进行去重，实现“*Exactly once*”语义。

如果业务数据产生消息可以找到合适的字段作为主键，或是有一个全局 ID 生成器，可以优先考虑选用第二种方案，这也是笔者在生产中选择的方案，如果读者有更好的解决方案，可考虑选用。

接下来讨论消费者部分。消费者处理消息与提交 offset 的顺序，在很大程度上决定了消息是哪个语义。在图 3-1 和图 3-2 中展示了两种提交 offset 不当导致的消息被重复消费（“*At least once*”语义）以及丢失消息（“*At most once*”语义）的情况。在图 3-1 的场景中，消费者拉取完消息后，业务逻辑先对消息进行处理，再提交 offset。这种模式下，如果消费者在处理完了消息之后，提交 offset 之前出现宕机，待消费者重新上线时，还会处理刚刚未提交的那部分消息（即 2 ~ 7 这部分消息），但这些消息已经被处理过了，这就对应于“*At least once*”语义。在图 3-2 的场景中，消费者拉取消息后，先提交 offset 后再处理消息。在提交 offset 之后，业务逻辑处理消息之前出现宕机，待消费者重新上线时，就无法读到刚刚已提交而未处理的这部分消息（即 5 ~ 8 这部分消息），这就对应于“*At most once*”语义。这里仅仅是一个示例，还有很多原因会导致类似的结果，例如 Consumer Group Rebalance 等。

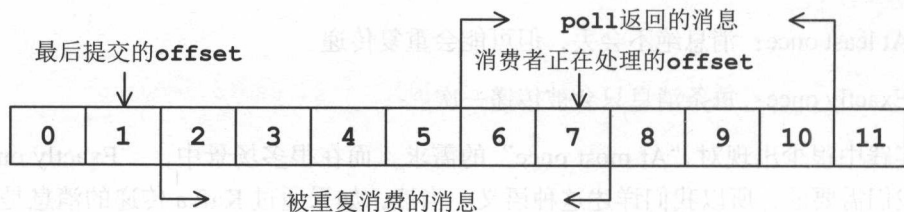


图 3-1

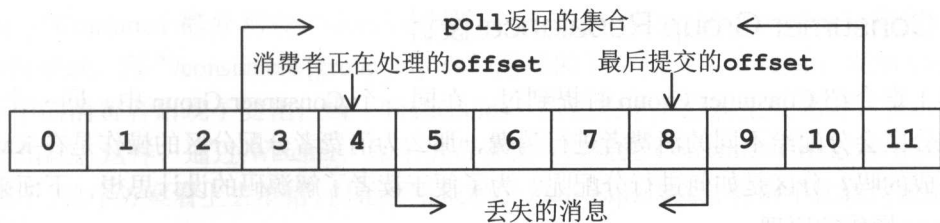


图 3-2

为了实现消费者的“Exactly once”语义，在这里提供一种方案，供读者参考：消费者将关闭自动提交 offset 的功能且不再手动提交 offset，这样就不使用 Offsets Topic 这个内部 Topic 记录其 offset，而是由消费者自己保存 offset。这里利用事务的原子性来实现“Exactly once”语义，我们将 offset 和消息处理结果放在一个事务中，事务执行成功则认为此消息被消费，否则事务回滚需要重新消费。当出现消费者宕机重启或 Rebalance 操作时，消费者可以从关系型数据库中找到对应的 offset，然后调用 `KafkaConsumer.seek()` 方法手动设置消费位置，从此 offset 处开始继续消费。

到目前为止，消费者并不知道 Consumer Group 什么时候会发生 Rebalance 操作，哪个分区分配给了哪个消费者消费。我们可以通过向 `KafkaConsumer` 添加 `ConsumerRebalanceListener` 接口来解决这个问题。`ConsumerRebalanceListener` 有两个回调方法。

- `onPartitionsRevoked()` 方法：调用时机是 Consumer 停止拉取数据之后、Rebalance 开始之前，我们可以在此方法中实现手动提交 offset，这就避免了 Rebalance 导致的重复消费的情况。
- `onPartitionsAssigned()` 方法：调用时机是 Rebalance 完成之后、Consumer 开始拉取数据之前，我们可以在此方法中调整或自定义 offset 的值。

通过 `ConsumerRebalanceListener` 接口和 `seek()` 方法，我们就可以实现从关系型数据库获取 offset 并手动设置的功能了。

这个方案还有其他的变体，例如，使用 `assign` 方法为消费者手动分配 `TopicPartition`，将提供事务保证的存储换成 HDFS 或其他 No-SQL 数据库，但是基本原理还是不变的。在笔者撰稿时，也有了实现“Exactly Once Delivery and Transactional Messaging”的相关意见 (<https://cwiki.apache.org/confluence/display/KAFKA/KIP-98>)，感兴趣的读者可以参考一下。

3.3 Consumer Group Rebalance 设计

第 1 章介绍 Consumer Group 时提到过，在同一个 Consumer Group 中，同一个 Topic 的不同分区会分配给不同的消费者进行消费，那么为消费者分配分区的操作是在 Kafka 服务端完成的吗？分区是如何进行分配呢？为了便于读者了解源码的设计思想，下面来分析 Rebalance 操作的原理。

方案一

Kafka 最开始的解决方案是通过 ZooKeeper 的 Watcher 实现的。每个 Consumer Group 在 ZooKeeper 下都维护了一个 “/consumers/[group_id]/ids” 路径，在此路径下使用临时节点记录属于此 Consumer Group 的消费者的 Id，由 Consumer 启动时创建。还有两个与 ids 节点同级的节点，它们分别是：owners 节点，记录了分区与消费者的对应关系；offsets 节点，记录了此 Consumer Group 在某个分区上的消费位置。

每个 Broker、Topic 以及分区在 ZooKeeper 中也都对应一个路径，如下所示。

- /brokers/ids/broker_id: 记录了 host、port 以及分配在此 Broker 上的 Topic 的分区列表。
- /brokers/topics/[topic_name]: 记录了每个 Partition 的 Leader、ISR 等信息。
- /brokers/topics/[topic_name]/partitions/[partition_num]/state: 记录了当前 Leader、选举 epoch 等信息。

路径图如图 3-3 所示。

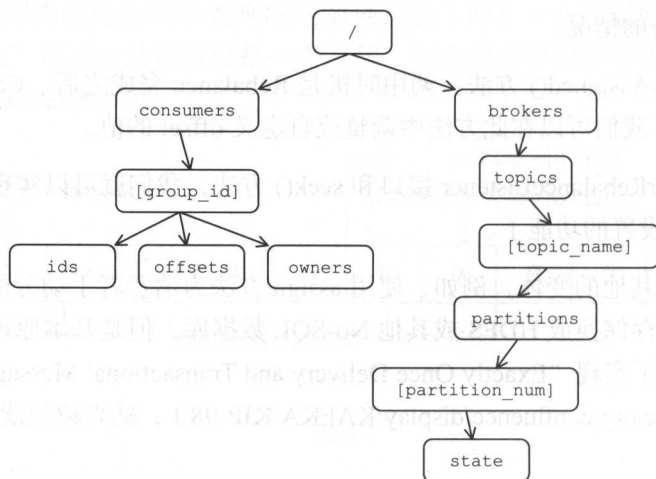


图 3-3

每个 Consumer 都分别在 “/consumers/[group_id]/ids” 和 “/brokers/ids” 路径上注册一个 Watcher。当 “/consumers/[group_id]/ids” 路径的子节点发生变化时，表示 Consumer Group 中的消费者出现了变化；当 “/brokers/ids” 路径的子节点发生变化时，表示 Broker 出现了增减。这样，通过 Watcher，每个消费者就可以监控 Consumer Group 和 Kafka 集群的状态了。这个方案看上去不错，但是严重依赖于 ZooKeeper 集群，有两个比较严重的问题：

- 羊群效应 (Herd Effect)：先解释一下什么是“羊群效应”，一个被 Watch 的 ZooKeeper 节点变化，导致大量的 Watcher 通知需要被发送给客户端，这将导致在通知期间其他操作延迟。一般出现这种情况的主要原因就是没有找到客户端真正的关注点，也算是滥用 Watcher 的一种场景。继续前面的分析，任何 Broker 或 Consumer 加入或退出，都会向其余所有的 Consumer 发送 Watcher 通知触发 Rebalance，就出现了“羊群效应”。
- 脑裂 (Split Brain)：每个 Consumer 都是通过 ZooKeeper 中保存的这些元数据判断 Consumer Group 状态、Broker 的状态以及 Rebalance 结果的，由于 ZooKeeper 只保证“最终一致性”，不保证“Simultaneously Consistent Cross-Client Views”，不同 Consumer 在同一时刻可能连接到 ZooKeeper 集群中不同的服务器，看到的元数据就可能不一样，这就会造成不正确的 Rebalance 尝试。ZooKeeper 的相关特性这里不再赘述，请读者参考资料进行学习。

方案二

由于上述两个原因，Kafka 的后续版本对 Rebalance 操作进行了改进，也对消费者进行了重新设计。其核心设计思想是：将全部的 Consumer Group 分成多个子集，每个 Consumer Group 子集在服务端对应一个 GroupCoordinator 对其进行管理，GroupCoordinator 是 KafkaServer 中用于管理 Consumer Group 的组件，其具体内容在第4章中详细介绍。消费者不再依赖 ZooKeeper，而只有 GroupCoordinator 在 ZooKeeper 上添加 Watcher。消费者在加入或退出 Consumer Group 时会修改 ZooKeeper 中保存的元数据，这点与上文描述的方案一类似，此时会触发 GroupCoordinator 设置的 Watcher，通知 GroupCoordinator 开始 Rebalance 操作。下面简述这个过程：

(1) 当前消费者准备加入某 Consumer Group 或是 GroupCoordinator 发生故障转移时，消费者并不知道 GroupCoordinator 的网络位置，消费者会向 Kafka 集群中的任一 Broker 发送 ConsumerMetadataRequest，此请求中包含了其 Consumer Group 的 GroupId，收到请求的 Broker 会返回 ConsumerMetadataResponse 作为响应，其中包含了管理此 Consumer Group 的 GroupCoordinator 的相关信息。

(2) 消费者根据 `ConsumerMetadataResponse` 中的 `GroupCoordinator` 信息, 连接到 `GroupCoordinator` 并周期性地发送 `HeartbeatRequest`, `HeartbeatRequest` 的具体格式在后面会详细介绍。发送 `HeartbeatRequest` 的主要作用是为了告诉 `GroupCoordinator` 此消费者正常在线, `GroupCoordinator` 会认为长时间未发送 `HeartbeatRequest` 的消费者已经下线, 触发新一轮的 `Rebalance` 操作。

(3) 如果 `HeartbeatResponse` 中带有 `IllegalGeneration` 异常, 说明 `GroupCoordinator` 发起了 `Rebalance` 操作, 此时消费者发送 `JoinGroupRequest` (具体格式在后面介绍) 给 `GroupCoordinator`, `JoinGroupRequest` 的主要目的是为了通知 `GroupCoordinator`, 当前消费者要加入指定的 `Consumer Group`。之后, `GroupCoordinator` 会根据收到的 `JoinGroupRequest` 和 `ZooKeeper` 中的元数据完成对此 `Consumer Group` 的分区分配。

(4) `GroupCoordinator` 会在分配完成后, 将分配结果写入 `ZooKeeper` 保存, 并通过 `JoinGroupResponse` 返回给消费者。消费者就可以根据 `JoinGroupResponse` 中分配的分区分开始消费数据。

(5) 消费者成功成为 `Consumer Group` 的成员后, 会周期性发送 `HeartbeatRequest`。如果 `HeartbeatResponse` 包含 `IllegalGeneration` 异常, 则执行步骤 3。如果找不到对应的 `GroupCoordinator` (`HeartbeatResponse` 包含 `NotCoordinatorForGroup` 异常), 则周期性地执行步骤 1, 直至成功。

这里只是简略地描述此方案的步骤, 整个方案还是有点复杂的, 其中比较严谨地描述了消费者和 `GroupCoordinator` 的状态图和各个阶段可能发生的故障以及故障转移处理, 本章重点关注 `Consumer Group Rebalance` 方面, 其他方面的内容在第 4 章中详细介绍。

上面这种方案虽然解决了“羊群效应”、“脑裂”问题, 但是还是有两个问题:

- 分区分配的操作是在服务端的 `GroupCoordinator` 中完成的, 这就要求服务端实现 `Partition` 的分配策略。当要使用新的 `Partition` 分配策略时, 就必须修改服务端的代码或配置, 之后重启服务, 这就显得比较麻烦。
- 不同的 `Rebalance` 策略有不同的验证需求。当需要自定义分区分配策略和验证需求时, 就会很麻烦。

方案三

为了解决上述问题, `Kafka 0.9` 版本中进行了重新设计, 将分区分配的工作放到了消费者这一端进行处理, 而 `Consumer Group` 管理的工作则依然由 `GroupCoordinator` 处理。这就让不同的模块关注不同的业务, 实现了业务的切分和解耦, 这种思想在设计时很重要。

下面简述一下重新设计后的协议。重新设计后的协议在上一版本的协议上进行了修改，将 JoinGroupRequest 的处理过程拆分成了两个阶段，分别是 Join Group 阶段和 Synchronizing Group State 阶段。

下面来看看新版本协议的变化。当消费者查找到管理当前 Consumer Group 的 GroupCoordinator 后，就会进入 Join Group 阶段，Consumer 首先向 GroupCoordinator 发送 JoinGroupRequest 请求，其中包含消费者的相关信息；服务端的 GroupCoordinator 收到 JoinGroupRequest 后会暂存消息，收集到全部消费者之后，根据 JoinGroupRequest 中的信息来确定 Consumer Group 中可用的消费者，从中选取一个消费者成为 Group Leader，还会选取使用的分区分配策略，最后将这些信息封装成 JoinGroupResponse 返回给消费者。

虽然每个消费者都会收到 JoinGroupResponse，但是只有 Group Leader 收到的 JoinGroupResponse 中封装了所有消费者的信息。当消费者确定自己是 Group Leader 后，会根据消费者的信息以及选定的分区分配策略进行分区分配。

在 Synchronizing Group State 阶段，每个消费者会发送 SyncGroupRequest 到 GroupCoordinator，但是只有 Group Leader 的 SyncGroupRequest 请求包含了分区的分配结果，GroupCoordinator 根据 Group Leader 的分区分配结果，形成 SyncGroupResponse 返回给所有 Consumer。消费者收到 SyncGroupResponse 后进行解析，即可获取分配给自身的分区。具体协议的细节在源码分析过程介绍。以上是新版本协议的核心变动，心跳检测、故障转移等方面并未改动，这里不再赘述。

最后，我们来了解消费者的状态转移与各请求之间的关系，如图 3-4 所示。

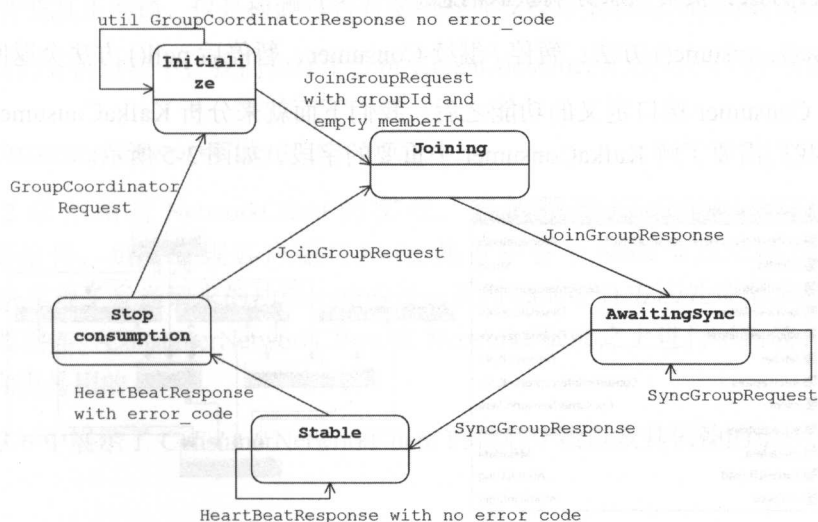


图 3-4

3.4 KafkaConsumer 分析

与 `KafkaProducer` 不同的是, `KafkaConsumer` 不是一个线程安全的类。为了便于分析,我们认为下面介绍的所有操作都是在同一线程中完成的,所以不需要考虑锁的问题。这种设计将实现多线程处理消息的逻辑转移到了调用 `KafkaConsumer` 的代码中,可以根据业务逻辑使用不同的实现方式。例如,可以使用“线程封闭”的方式,每个业务线程拥有一个 `KafkaConsumer` 对象,这种方式实现简单、快速。还可以使用两个线程池实现“生产者—消费者”模式,解耦消息消费和消息处理的逻辑。其中一个线程池中每个线程拥有一个 `KafkaConsumer` 对象,负责从 `Kafka` 集群拉取消息,然后将消息放入队列中缓存,而另一个线程池中的线程负责从队列中获取消息,执行处理消息的业务逻辑。读者可以根据自己的工作场景,提出更为灵活的方案。

下面开始对 `KafkaConsumer` 的分析。`KafkaConsumer` 实现了 `Consumer` 接口, `Consumer` 接口中定义了 `KafkaConsumer` 对外的 API,其核心方法可以分为下面六类。

- `subscribe()` 方法: 订阅指定的 Topic, 并为消费者自动分配分区。
- `assign()` 方法: 用户手动订阅指定的 Topic, 并且指定消费的分区。此方法与 `subscribe()` 方法互斥, 在后面会详细介绍是如何实现互斥的。
- `commit*()` 方法: 提交消费者已经消费完成的 offset。
- `seek*()` 方法: 指定消费者起始消费的位置。
- `poll()` 方法: 负责从服务端获取消息。
- `pause()`、`resume()` 方法: 暂停 / 继续 Consumer, 暂停后 `poll()` 方法会返回空。

了解了 `Consumer` 接口定义的功能之后, 我们下面就来分析 `KafkaConsumer` 的具体实现。首先, 我们需要了解 `KafkaConsumer` 中重要的字段, 如图 3-5 所示。

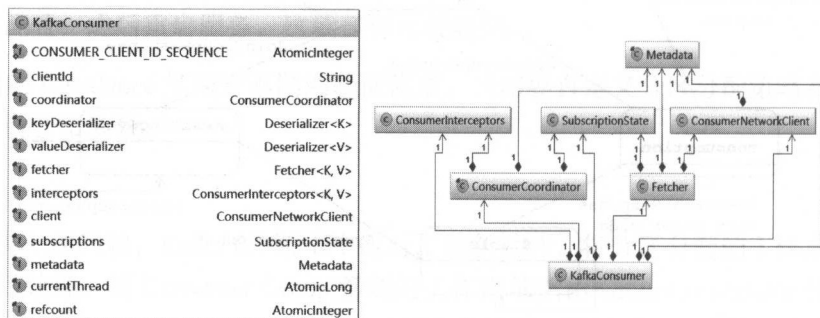


图 3-5

- `PRODUCER_CLIENT_ID_SEQUENCE`: `clientId` 的生成器, 如果没有明确指定 `client` 的 `Id`, 则使用字段生成一个 `ID`。
- `clientId`: `Consumer` 的唯一标示。
- `coordinator`: 控制着 `Consumer` 与服务端 `GroupCoordinator` 之间的通信逻辑, 读者可以将其理解成 `Consumer` 与服务端 `GroupCoordinator` 通信的门面。
- `keyDeserializer` 和 `valueDeserializer`: `key` 反序列化器和 `value` 反序列化器, 参见第2章相关章节。
- `fetcher`: 负责从服务端获取消息。
- `interceptors`: `ConsumerInterceptor` 集合, `ConsumerInterceptor.onConsumer()` 方法可以在消息通过 `poll()` 方法返回给用户之前对其进行拦截或修改; `ConsumerInterceptor.onCommit()` 方法也可以在服务端返回提交 `offset` 成功的响应时对其进行拦截或修改。与第2章介绍的 `ProducerInterceptors` 类似。
- `client`: 负责消费者与 `Kafka` 服务端的网络通信。
- `subscriptions`: 维护了消费者的消费状态。
- `metadata`: 记录了整个 `Kafka` 集群的元信息。
- `currentThread` 和 `refcount`: 分别记录了当前使用 `KafkaConsumer` 的线程 `Id` 和重入次数, `KafkaConsumer` 的 `acquire()` 方法和 `release()` 方法实现了一个“轻量级锁”, 它并非真正的锁, 仅是检测是否有多线程并发操作 `KafkaConsumer` 而已。

在后面的分析过程中, 我们会逐个分析 `KafkaConsumer` 依赖的组件的功能和实现。

3.4.1 ConsumerNetworkClient

在第2章介绍过 `NetworkClient` 的实现, 它依赖于 `KSelector`、`InFlightRequests`、`Metadata` 等组件, 负责管理客户端与 `Kafka` 集群中各个 `Node` 节点之间的连接, 通过 `KSelector` 法实现了发送请求的功能, 并通过一系列 `handle*()` 方法处理请求响应、超时请求以及断线重连。`ConsumerNetworkClient` 在 `NetworkClient` 之上进行了封装, 提供了更高级的功能和更易用的 `API`。

在图 3-6 中展示了 `ConsumerNetworkClient` 的核心字段以及其依赖的组件。

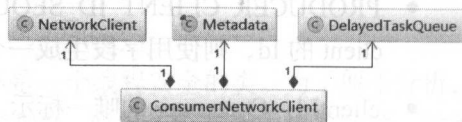
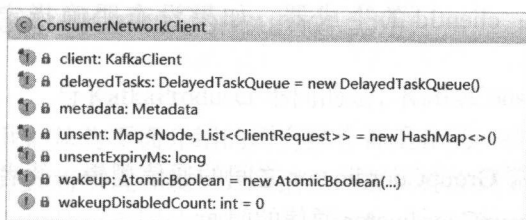


图 3-6

- client: NetworkClient 对象。
- delayedTasks: 定时任务队列, DelayedTaskQueue 是 Kafka 提供的定时任务队列的实现, 其底层是使用 JDK 提供的 PriorityQueue 实现。简单介绍一下 PriorityQueue, 这是一个非线程安全的、无界的、优先级队列, 实现原理是小顶堆, 底层是基于数组实现的, 其对应的线程安全实现是 PriorityBlockingQueue。有兴趣的读者可以参考 JDK 相关源码。在后面的分析中我们可以看到, 这个定时任务队列中是心跳任务。
- metadata: 用于管理 Kafka 集群元数据。
- unsent: 缓冲队列, Map<Node, List<ClientRequest>> 类型, key 是 Node 节点, value 是发往此 Node 的 ClientRequest 集合。
- unsentExpiryMs: ClientRequest 在 unsent 中缓存的超时时长。
- wakeup: 由调用 KafkaConsumer 对象的消费者线程之外的其他线程设置, 表示要中断 KafkaConsumer 线程。
- wakeupDisabledCount: KafkaConsumer 是否正在执行不可中断的方法。每进入一个不可中断的方法时, 则增加一, 退出不可中断方法时, 则减少一。wakeupDisabledCount 只会被 KafkaConsumer 线程修改, 其他线程不能修改。

ConsumerNetworkClient.poll() 方法是 ConsumerNetworkClient 中最核心的方法, poll() 方法有多个重载, 最终会调用 poll(long timeout, long now, boolean executeDelayedTasks) 重载, 这三个参数的含义分别是: timeout 表示执行 poll() 方法的最长阻塞时间 (单位是 ms), 如果为 0, 则表示不阻塞; now 表示当前时间戳; executeDelayedTasks 表示是否执行 delayedTasks 队列中的定时任务。下面介绍其流程, 其中简单回顾一下 NetworkClient 的功能:

- (1) 调用 ConsumerNetworkClient.trySend() 方法循环处理 unsent 中缓存的请求。具

体逻辑是：对每个 Node 节点，循环遍历其对应的 ClientRequest 列表，每次循环都调用 NetworkClient.ready() 方法检测消费者与此节点之间的连接，以及发送请求的条件。若符合发送条件，则调用 NetworkClient.send() 方法将请求放入 InFlightRequests 队中等待响应，也放入 KafkaChannel 的 send 字段中等待发送，并将此消息从列表中删除。实现代码如下：

```
private boolean trySend(long now) {
    boolean requestsSent = false;
    for (Map.Entry<Node, List<ClientRequest>> requestEntry : unsent
        .entrySet()) { // 遍历 unsent 集合
        Node node = requestEntry.getKey();
        Iterator<ClientRequest> iterator = requestEntry.getValue().
iterator();
        while (iterator.hasNext()) {
            ClientRequest request = iterator.next();
            // 调用 NetworkClient.ready() 检测是否可以发送请求
            if (client.ready(node, now)) {
                // 调用 NetworkClient.send() 方法，等待发送请求
                client.send(request, now);
                iterator.remove(); // 从 unsent 集合中删除此请求
                requestsSent = true;
            }
        }
    }
    return requestsSent;
}
```

(2) 计算超时时间，此超时时间由 timeout 与 delayedTasks 队列中最近要执行的定时任务的时间共同决定。在下面的 NetworkClient.poll() 方法中，会使用此超时时间作为最长阻塞时长，避免影响定时任务的执行。

(3) 调用 NetworkClient.poll() 方法，将 KafkaChannel.send 字段指定的消息发送出去。除此之外，NetworkClient.poll() 方法可能会更新 Metadata 使用一系列 handle*() 方法处理请求响应、连接断开、超时等情况，并调用每个请求的回调函数。NetworkClient 的详细实现参考第 2 章。

(4) 调用 ConsumerNetworkClient.maybeTriggerWakeup() 方法，检测 wakeup 和 wakeupDisabledCount，查看是否有其他线程中断。如果有中断请求，则抛出 WakeupException 异常，中断当前 ConsumerNetworkClient.poll() 方法。


```
private void maybeTriggerWakeup() {
    // 通过 wakeupDisabledCount 检测是否在执行不可中断的方法, 通过 wakeup 检测是否有中断请求
    if (wakeupDisabledCount == 0 && wakeup.get()) {
        wakeup.set(false); // 重置中断标志
        throw new WakeupException();
    }
}
```

(5) 调用 `checkDisconnects()` 方法检测连接状态。检测消费者与每个 Node 之间的连接状态, 当检测到连接断开的 Node 时, 会将其在 `unsent` 集合中对应的全部 `ClientRequest` 对象清除掉, 之后调用这些 `ClientRequest` 的回调函数。

```
private void checkDisconnects(long now) {
    Iterator<Map.Entry<Node, List<ClientRequest>>> iterator =
        unsent.entrySet().iterator();
    while (iterator.hasNext()) { // 遍历 unsent 集合中的每个 Node
        Map.Entry<Node, List<ClientRequest>> requestEntry = iterator.next();
        Node node = requestEntry.getKey();
        if (client.connectionFailed(node)) { // 检测消费者与每个 Node 之间的连接状态
            iterator.remove(); // 从 unsent 集合中删除此 Node 对应的全部 ClientRequest
            for (ClientRequest request : requestEntry.getValue()) {
                RequestFutureCompletionHandler handler =
                    (RequestFutureCompletionHandler) request.callback();
                // 调用 ClientRequest 的回调函数
                handler.onComplete(new ClientResponse(request, now, true, null));
            }
        }
    }
}
```

断开连接的 Node 所对应的已经发送出去的请求, 由 `NetworkClient` 进行异常处理, 具体实现参考第 2 章。

(6) 根据 `executeDelayedTasks` 参数决定是否处理 `delayedTasks` 队列中超时的定时任务, 如果需要执行 `delayedTasks` 队列中的定时任务, 则调用 `delayedTasks.poll()` 方法。

(7) 再次调用 `trySend()` 方法。在步骤 3 中调用了 `NetworkClient.poll()` 方法, 在其中可能已经将 `KafkaChannel.send` 字段上的请求发送出去了, 也可能已经新建了与某些 Node 的网络连接, 所以这里再次尝试调用 `trySend()` 方法。

(8) 调用 `ConsumerNetworkClient.failExpiredRequests()` 处理 `unsent` 中超时请求。它会循环遍历整个 `unsent` 集合, 检测每个 `ClientRequest` 是否超时, 调用超时 `ClientRequest` 的回调函数, 并将其从 `unsent` 集合中删除。

```
private void failExpiredRequests(long now) {
    Iterator<Map.Entry<Node, List<ClientRequest>>> iterator = unsent
        .entrySet().iterator();
    while (iterator.hasNext()) { // 遍历 unsent 集合
        ... .. // 创建迭代器 (略)
        while (requestIterator.hasNext()) {
            ClientRequest request = requestIterator.next();
            if (request.createdTimeMs() < now - unsentExpiryMs) { // 检查是否超时
                RequestFutureCompletionHandler handler =
                    (RequestFutureCompletionHandler) request.callback();
                handler.raise(new TimeoutException("...")); // 调用回调函数
                requestIterator.remove(); // 删除 ClientRequest
            } else
                break;
        }
        if (requestEntry.getValue().isEmpty()) // 队列已经为空, 则从 unsent 集合中删除
            iterator.remove();
    }
}
```

分析完 `poll()` 方法的详细步骤之后, 我们下面来看其实现代码:

```
private void poll(long timeout, long now, boolean executeDelayedTasks) {
    trySend(now); // 步骤 1: 检测发送条件, 将请求放入 KafkaChannel.send 字段, 待发送
    ... ..
    // 步骤 2: 计算超时时间
    timeout = Math.min(timeout, delayedTasks.nextTimeout(now));
    ... ..
    // 步骤 3、4: 调用 NetworkClient.poll() 方法, 并检测是否有中断请求
    clientPoll(timeout, now);
    now = time.milliseconds(); // 重置当前时间
    checkDisconnects(now); // 步骤 5: 根据连接状态, 处理 unsent 中的请求
    if (executeDelayedTasks) // 步骤 6: 处理定时任务
        delayedTasks.poll(now);
    // 步骤 7: 检测发送条件, 重新设置 KafkaChannel.send 字段, 并超时断线重连
}
```

```
trySend(now);
failExpiredRequests(now); // 步骤 8: 处理 unsent 中的超时任务
}
```

`pollNoWakeup()` 方法是 `poll()` 方法的变体, 表示执行不可被中断的 `poll()` 方法。具体逻辑是: 在执行 `poll()` 方法之前, 会调用 `disableWakeup()` 方法将 `wakeupDisabledCount` 加一, 然后调用 `poll()` 方法。这样, 即使其他线程请求中断, 也不会被响应。

`poll(future)` 是 `poll()` 方法的另一个实现阻塞发送请求的功能, 代码如下所示。

```
public void poll(RequestFuture<?> future) {
    while (!future.isDone()) { // 循环检测 future, 即请求的完成情况
        poll(Long.MAX_VALUE); // 请求未完成, 则调用 poll() 方法
    }
}
```

在 `ConsumerNetworkClient.send()` 方法中, 会将待发送的请求封装成 `ClientRequest`, 然后保存到 `unsent` 集合中等待发送, 具体代码如下。

```
public RequestFuture<ClientResponse> send(Node node, ApiKeys api,
    AbstractRequest request) {
    long now = time.milliseconds();
    RequestFutureCompletionHandler future = new RequestFutureCompletionHand
ler();
    RequestHeader header = client.nextRequestHeader(api);
    RequestSend send = new RequestSend(node.idString(), header, request.
toStruct());
    // 创建 ClientRequest 对象, 并保存到 unsent 集合中
    put(node, new ClientRequest(now, true, send, future));
    return future;
}
```

在这里需要重点关注的是 `KafkaConsumer` 中使用的回调对象——`RequestFutureCompletionHandler`, 其继承关系如图 3-7 所示。`RequestCompleteHandler` 接口只有 `onComplete()` 方法, 此接口的另一个实现是第 2 章中介绍的 `KafkaProducer` 端的请求回调 (`Sender` 的一个匿名内部类)。

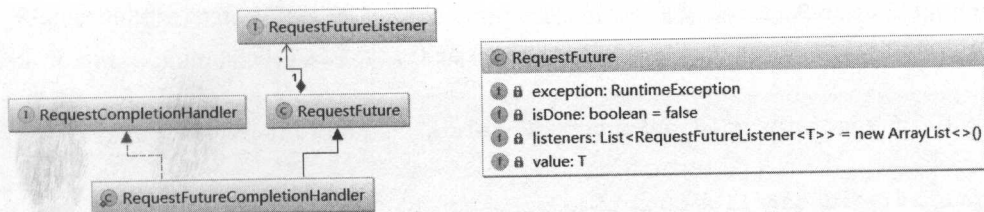


图 3-7

RequestFutureCompletionHandler.onComplete() 方法的代码如下：

```

public void onComplete(ClientResponse response) {
    if (response.wasDisconnected()) { // 因连接故障而产生的 ClientResponse 对象
        ...
        // 调用继承自父类 RequestFuture 的 raise() 方法
        raise(DisconnectException.INSTANCE);
    } else {
        complete(response); // 调用继承自父类 RequestFuture 的 complete() 方法
    }
}

```

从 RequestFutureCompletionHandler 的继承关系上我们可以知道，它不仅实现了 RequestCompletionHandler，它还继承了 RequestFuture 类。RequestFuture 是一个泛型类，其核心字段如下所示。

- **isDone**：表示当前请求是否已经完成，不管正常完成还是出现异常，此字段都会被设置为 true。
- **exception**：记录导致请求异常完成的异常类，与 value 字段互斥。此字段非空则表示出现异常，反之则表示正常完成。
- **value**：记录请求正常完成时收到的响应，与 exception 字段互斥。此字段非空表示正常完成，反之则表示出现异常。
- **listeners**：RequestFutureListener 集合，用来监听请求完成的情况。RequestFutureListener 接口有 onSuccess() 和 onFailure() 两个方法，对应于请求正常完成和出现异常两种情况。

在 RequestFuture 中有两处典型设计模式的使用：一处是 compose() 方法，使用了适配器模式；另一处是 chain() 方法，使用了责任链模式。下面是 compose() 方法的相关代码：

```
// RequestFutureAdapter 就是一个适配器
public abstract class RequestFutureAdapter<F, T> {

    public abstract void onSuccess(F value, RequestFuture<T> future);

    public void onFailure(RuntimeException e, RequestFuture<T> future) {
        future.raise(e);
    }
}

// 下面是 RequestFuture.compose() 方法的实现, 它将 RequestFuture<T> 适配成
RequestFuture<S>
public <S> RequestFuture<S> compose(final RequestFutureAdapter<T, S>
adapter) {
    final RequestFuture<S> adapted = new RequestFuture<S>(); // 适配之后的结果
    // 在当前 RequestFuture 上添加监听器
    addListener(new RequestFutureListener<T>() {
        public void onSuccess(T value) {
            adapter.onSuccess(value, adapted);
        }
        public void onFailure(RuntimeException e) {
            adapter.onFailure(e, adapted);
        }
    });
    return adapted;
}
```

图 3-8 展示了使用 `compose()` 方法进行适配后, 回调时的调用过程, 也可以认为是请求完成的事件传播流程。当调用 `RequestFuture<T>` 对象的 `complete()` 或 `raise()` 方法时, 会调用 `RequestFutureListener<T>` 的 `onSuccess()` 或 `onFailure()` 方法, 然后调用 `RequestFutureAdapter<T,S>` 的对应方法, 最终调用 `RequestFuture<S>` 对象的对应方法。

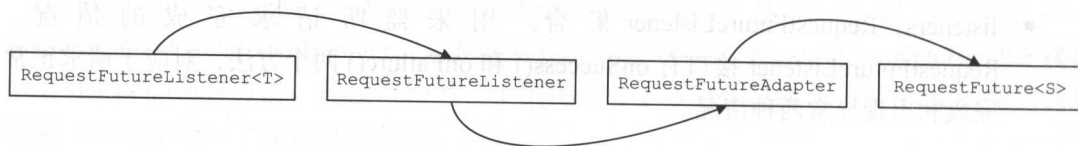


图 3-8

`RequestFuture.chain()` 方法的实现与 `compose()` 类似，也是通过 `RequestFutureListener` 在多个 `RequestFuture` 之间传递事件。下面是其具体代码：

```
public void chain(final RequestFuture<T> future) {
    addListener(new RequestFutureListener<T>() { // 添加监听器
        public void onSuccess(T value) {
            // 通过监听器将 value 传递给下一个 RequestFuture 对象
            future.complete(value);
        }
        public void onFailure(RuntimeException e) {
            future.raise(e); // 通过监听器将异常传递给下一个 RequestFuture 对象
        }
    });
}
```

`RequestFuture` 提供了一系列检查请求完成情况的方法，以及管理 `listeners` 的方法，代码比较简单，不再赘述了。

介绍完 `RequestFutureCompleteHandler` 之后，回到 `ConsumerNetworkClient` 的分析上来。下面简单介绍 `ConsumerNetworkClient` 中几个常用的功能，代码比较简单，就不贴出来了：

- `awaitMetadataUpdate()` 方法：循环调用 `poll()` 方法，直到 `Metadata` 版本号增加，实现阻塞等待 `Metadata` 更新完成。
- `awaitPendingRequests()` 方法：等待 `unsent` 和 `InFlightRequests` 中的请求全部完成（正常收到响应或出现异常）。
- `put()` 方法：向 `unsent` 中添加请求。
- `schedule()` 方法：向 `delayedTasks` 队列中添加定时任务。
- `leastLoadedNode()` 方法：查找 `Kafka` 集群中负载最低的 `Node`。

3.4.2 SubscriptionState

`KafkaConsumer` 从 `Kafka` 拉取消息时发送的请求是 `FetchRequest`（具体格式后面介绍），在其中需要指定消费者希望拉取的起始消息的 `offset`。为了消费者快速获取这个值，`KafkaConsumer` 使用 `SubscriptionState` 来追踪 `TopicPartition` 与 `offset` 对应关系。图 3-9 展示了 `SubscriptionState` 依赖的类以及其核心字段。

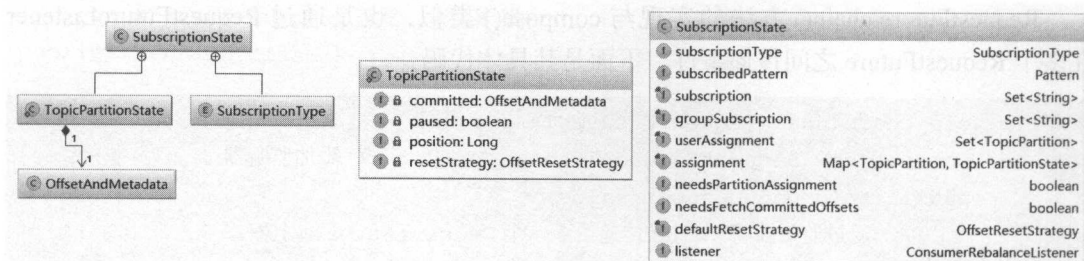


图 3-9

`SubscriptionType` 是 `SubscriptionState` 的一个内部枚举类型, 表示的是订阅 Topic 的模式, 分为四类。

- `NONE`: `SubscriptionState.subscriptionType` 的初始值。
- `AUTO_TOPICS`: 按照指定的 Topic 名字进行订阅, 自动分配分区。
- `AUTO_PATTERN`: 按照指定的正则表达式匹配 Topic 进行订阅, 自动分配分区。
- `USER_ASSIGNED`: 用户手动指定消费者消费的 Topic 以及分区编号。

`TopicPartitionState` 表示的是 `TopicPartition` 的消费状态, 其关键字段如下所示。

- `position`: 记录了下次要从 Kafka 服务端获取的消息的 offset。
- `committed`: 记录了最近一次提交的 offset。
- `paused`: 记录了当前 `TopicPartition` 是否处于暂停状态, 与 `Consumer` 接口的 `pause()` 方法相关。
- `resetStrategy`: `OffsetResetStrategy` 枚举类型, 重置 `position` 的策略。同时, 此字段是否为空, 也表示了是否需要重置 `position` 的值。

`TopicPartitionState` 提供了管理上面四个字段方法, 比较简单, 不再赘述。

在前面介绍 `Consumer` 接口时提到过, `subscribe()` 方法和 `assign()` 方法是互斥的。其实上面介绍的三种模式都是互斥的。下面是 `setSubscriptionType()` 方法的代码, 无论选择哪种模式都会调用此方法进行设置, 如图 3-10 所示。

```

SubscriptionState.setSubscriptionType(SubscriptionType) (org.apache.kafka.clients.consumer.internals)
  ▶ SubscriptionState.subscribe(Collection<String>, ConsumerRebalanceListener) (org.apache.kafka.clients.consumer.internals)
  ▶ SubscriptionState.subscribe(Pattern, ConsumerRebalanceListener) (org.apache.kafka.clients.consumer.internals)
  ▶ SubscriptionState.assignFromUser(Collection<TopicPartition>) (org.apache.kafka.clients.consumer.internals)
  
```

图 3-10

```
private void setSubscriptionType(SubscriptionType type) {
    // 如果是 NONE, 则可以指定其他模式
    if (this.subscriptionType == SubscriptionType.NONE)
        this.subscriptionType = type;
    else if (this.subscriptionType != type) // 如果已经指定了其他模式, 则会报错
        throw new IllegalStateException(SUBSCRIPTION_EXCEPTION_MESSAGE);
}
```

下面介绍 SubscriptionState 的核心字段。

- subscriptionType: SubscriptionType 枚举类型, 表示订阅的模式。
- subscribedPattern: 使用 AUTO_PATTERN 模式时, 是按照此字段记录的正则表达式对所有 Topic 进行匹配, 对匹配符合的 Topic 进行订阅。
- subscription: 如果使用 AUTO_TOPICS 或 AUTO_PATTERN 模式, 则使用此集合记录所有订阅的 Topic。向 subscription 集合中添加数据的方法只有 changeSubscription() 方法, 而调用 changeSubscription() 方法有两处, 如图 3-11 所示。

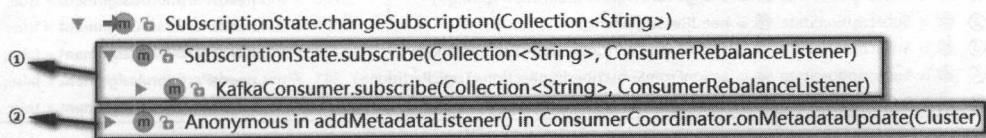


图 3-11

在图 3-11 中的①处, 使用的是 AUTO_TOPICS 模式订阅; 图 3-11 中的②处使用 AUTO_PATTERN 模式订阅。我们在前面介绍 Metadata 的时候提到过, 可以在其上添加 Listener, 当 Metadata 更新时会触发 `Metadata.Listener.onMetadataUpdate()` 方法, 图 3-11 中的②处就是在 Metadata 的 Listener 中通过 `subscribedPattern` 模式过滤 Topic, 并调用 `changeSubscription()` 方法修改 `subscription` 集合。

- userAssignment: 如果使用 USER_ASSIGNED 模式, 则此集合记录了分配给当前消费者的 TopicPartition 集合。SubscriptionType 模式是互斥的, 所以 userAssignment 集合与 subscription 集合也是互斥的。
- assignment: Map<TopicPartition, TopicPartitionState> 类型, 无论使用什么订阅模式, 都使用此集合记录每个 TopicPartition 的消费状态。
- groupSubscription: 在前面描述的协议中, Consumer Group 中会选举一个

Leader, Leader 使用该集合记录 Consumer Group 中所有消费者订阅的 Topic, 而其他 Follower 的该集合中只保存了其自身的订阅的 Topic。图 3-12 是与 groupSubscription 集合相关的方法。

①	SubscriptionState	changeSubscription(Collection<String>)	175	↪this.groupSubscription.addAll(topicsToSubscribe);
②	SubscriptionState	groupSubscribe(Collection<String>)	199	↪this.groupSubscription.addAll(topics);
③	SubscriptionState	needReassignment()	203	↪this.groupSubscription.retainAll(subscription);

图 3-12

图 3-12 中的①处是将消费者自身订阅的 Topic 添加到 groupSubscribe 集合; ②处是在 Leader 收到 JoinGroupResponse 时调用, 在 JoinGroupResponse 中包含了全部消费者订阅的 Topic, 在此时将 Topic 信息添加到 groupSubscribe 集合。③处则是将 groupSubscribe 中其他消费者订阅的 Topic 删除, 只留下自身订阅的 Topic (即 subscription 集合), 这是 groupSubscription 集合收缩的场景。

- needsPartitionAssignment: 标记是否需要进行一次分区分配。这里简单了解一下修改 needPartitionAssignment 的场景和含义, 如图 3-13 所示。

①	SubscriptionState	changeSubscription(Collection<String>)	176	↪this.needsPartitionAssignment = true;
②	SubscriptionState	needReassignment()	204	↪this.needsPartitionAssignment = true;
③	SubscriptionState	assignFromUser(Collection<TopicPartition>)	224	↪this.needsPartitionAssignment = false;
④	SubscriptionState	assignFromSubscribed(Collection<TopicPartition>)	242	↪this.needsPartitionAssignment = false;
⑤	SubscriptionState	unsubscribe()	264	↪this.needsPartitionAssignment = true;

图 3-13

图 3-13 中的①、⑤处将 needsPartitionAssignment 设置为 true 是因为消费者订阅的 Topic 发生了变化, 所以需要进行分区分配; ③处将 needsPartitionAssignment 设置为 false 是因为使用 USER_ASSIGNED 订阅模式, 所以不需要分区分配操作; ④处是成功得到 SyncGroupResponse 中的分区分配结果时的操作, 此时 Rebalance 操作结束, 将 needsPartitionAssignment 设置为 false; ②处的场景比较复杂, 调用②处将 needReassignment 设置为 true, 主要是因为某些请求响应中出现了 ILLEGAL_GENERATION 等异常, 或是订阅的 Topic 出现了分区数量的变化, 调用关系如图 3-14 所示。

SubscriptionState.needReassignment()
▶ OffsetFetchResponseHandler in ConsumerCoordinator.handle(OffsetFetchResponse, RequestFuture<Map<TopicPartition, OffsetAndMetadata>>)
▶ Anonymous in addMetadataListener() in ConsumerCoordinator.onMetadataUpdate(Cluster)
▶ OffsetCommitResponseHandler in ConsumerCoordinator.handle(OffsetCommitResponse, RequestFuture<Void>)
▶ ConsumerCoordinator.onJoinComplete(int, String, String, ByteBuffer)

图 3-14

- needsFetchCommittedOffsets: 标记是否需要从 GroupCoordinator 获取最近提交的

offset。当出现异步提交 offset 操作或是 Rebalance 操作刚完成时会将其置为 true，成功获取最近提交 offset 之后会设置为 false。

- defaultResetStrategy: 默认 OffsetResetStrategy 策略。
- listener: ConsumerRebalanceListener 类型，用于监听分区分配操作，在前面介绍过，不再赘述。

SubscriptionState 中的方法主要是管理上面的几个集合字段，操作比较简单，不再详细介绍。下面简单分析前面示例中使用的 subscribe() 方法：

```
public void subscribe(Collection<String> topics, ConsumerRebalanceListener
listener) {
    // 用户未指定 ConsumerRebalanceListener 时，默认使用 NoOpConsumerRebalanceListener
    // 其中所有方法的实现都是空的
    if (listener == null)
        throw new IllegalArgumentException("...");
    setSubscriptionType(SubscriptionType.AUTO_TOPICS); // 选择 AUTO_TOPICS 模式
    this.listener = listener;
    changeSubscription(topics);
}

public void changeSubscription(Collection<String> topicsToSubscribe) {
    // 订阅的 Topic 有变化
    if (!this.subscription.equals(new HashSet<>(topicsToSubscribe))) {
        this.subscription.clear(); // 清空 subscription 集合
        this.subscription.addAll(topicsToSubscribe); // 添加订阅的 Topic
        this.groupSubscription.addAll(topicsToSubscribe);
        this.needsPartitionAssignment = true; // 标记需要重新分配分区
        // 同步 assignment 与 subscription 集合
        for (Iterator<TopicPartition> it =
            assignment.keySet().iterator(); it.hasNext();) {
            TopicPartition tp = it.next();
            if (!subscription.contains(tp.topic()))
                it.remove();
        }
    }
}
```

当 Topic 得到需要订阅的 Topic 集合后，设置到 SubscriptionState。

3.4.3 ConsumerCoordinator

在前面介绍了 Kafka 中 Rebalance 操作的相关方案和原理。在 KafkaConsumer 中通过 ConsumerCoordinator 组件实现与服务端的 GroupCoordinator 的交互，ConsumerCoordinator 继承了 AbstractCoordinator 抽象类。下面我们先来介绍 AbstractCoordinator 的核心字段，如图 3-15 所示。

AbstractCoordinator	
heartbeat	Heartbeat
heartbeatTask	HeartbeatTask
groupId	String
client	ConsumerNetworkClient
needsJoinPrepare	boolean
rejoinNeeded	boolean
coordinator	Node
memberId	String
generation	int

ConsumerCoordinator	
assignmentSnapshot	MetadataSnapshot
assignors	List<PartitionAssignor>
autoCommitEnabled	boolean
autoCommitTask	AutoCommitTask
defaultOffsetCommitCallback	OffsetCommitCallback
excludeInternalTopics	boolean
interceptors	ConsumerInterceptors<?, ?>
metadata	Metadata
metadataSnapshot	MetadataSnapshot
subscriptions	SubscriptionState

图 3-15

- **heartbeat**: 心跳任务的辅助类，其中记录了两发发送心跳消息的间隔（interval 字段）、最近发送心跳的时间（lastHeartbeatSend 字段）、最后收到心跳响应的时间（lastHeartbeatReceive 字段）、过期时间（timeout 字段）、心跳任务重置时间（lastSessionReset 字段），同时还提供了计算下次发送心跳的时间（timeToNextHeartbeat() 方法）、检测是否过期的方法（sessionTimeoutExpired() 方法）。
- **heartbeatTask**: HeartbeatTask 是一个定时任务，负责定时发送心跳请求和心跳响应的处理，会被添加到前面介绍的 ConsumerNetworkClient.delayedTasks 定时任务队列中。
- **groupId**: 当前消费者所属的 Consumer Group 的 Id。
- **client**: ConsumerNetworkClient 对象，负责网络通信和执行定时任务。
- **needsJoinPrepare**: 标记是否需要执行发送 JoinGroupRequest 请求前的准备操作。
- **rejoinNeeded**: 此字段是否重新发送 JoinGroupRequest 请求的条件之一。下面先简单了解修改其值的地方和含义，如图 3-16 所示。

①	AbstractCoordinator.JoinGroupResponseHandler	handle(JoinGroupResponse, RequestFuture<ByteBuffer>)	409	◁AbstractCoordinator.this.rejoinNeeded = false;
②	AbstractCoordinator.SyncGroupResponseHandler	handle(SyncGroupResponse, RequestFuture<ByteBuffer>)	535	◁AbstractCoordinator.this.rejoinNeeded = true;
③	AbstractCoordinator	maybeLeaveGroup()	700	◁rejoinNeeded = true;
④	AbstractCoordinator.HeartbeatCompletionHandler	handle(HeartbeatResponse, RequestFuture<Void>)	781	◁AbstractCoordinator.this.rejoinNeeded = true;

图 3-16

上图①处是收到正常的 JoinGroupResponse 响应，将 rejoinNeeded 设置为 false，防止重复发送 JoinGroupRequest 请求。②、③、④三处分别是收到异常的 SyncGroupResponse 或 HeartbeatResponse 或消费者离开 Consumer Group 时执行的操作，它们都会将 rejoinNeeded 设置为 true，表示可以重新发送 JoinGroupRequest。

- coordinator: Node 类型，记录服务端 GroupCoordinator 所在的 Node 节点。
- memberId: 服务端 GroupCoordinator 返回的分配给消费者的唯一 Id。
- generation: 服务端 GroupCoordinator 返回的年代信息，用来区分两次 Rebalance 操作。由于网络延迟等问题，在执行 Rebalance 操作时可能收到上次 Rebalance 过程的请求，避免这种干扰，每次 Rebalance 操作都会递增 generation 的值。

下面是 ConsumerCoordinator 的核心字段。

- assignors: PartitionAssignor 列表。在消费者发送的 JoinGroupRequest 请求中包含了消费者自身支持的 PartitionAssignor 信息，GroupCoordinator 从所有消费者都支持的分配策略中选择一个，通知 Leader 使用此分配策略进行分区分配。此字段的值通过 partition.assignment.strategy 参数配置，可以配置多个。
- metadata: 记录了 Kafka 集群的元数据。
- subscriptions: SubscriptionState 对象，参考 SubscriptionState 小节。
- autoCommitEnabled: 是否开启了自动提交 offset。
- autoCommitTask: 自动提交 offset 的定时任务。
- interceptors: ConsumerInterceptor 集合，在前面介绍过，不再赘述。
- excludeInternalTopics: 标识是否排除内部 Topic。
- metadataSnapshot: 用来存储 Metadata 的快照信息，主要用来检测 Topic 是否发生了分区数量的变化。在 ConsumerCoordinator 的构造方法中，会为 Metadata 添加一个监听器，当 Metadata 更新时会做下面几件事。

◀ 如果是 AUTO_PATTERN 模式，则使用用户自定义的正则表达式过滤 Topic，得到需要订阅的 Topic 集合后，设置到 SubscriptionState 的

subscription 集合和 groupSubscription 集合中。

- ◀ 如果是 AUTO_PATTERN 或 AUTO_TOPICS 模式，为当前 Metadata 做一个快照，这个快照底层是使用 HashMap 记录每个 Topic 中 Partition 的个数。将新旧快照进行比较，发生变化的话，则表示消费者订阅的 Topic 发生分区数量变化，则将 SubscriptionState 的 needsPartitionAssignment 字段置为 true，需要重新进行分区分配。
- ◀ 使用 metadataSnapshot 字段记录变化后的新快照。

我们下面来分析 Metadata 的 Listener 的具体实现：

```
private void addMetadataListener() {
    this.metadata.addListener(new Metadata.Listener() {
        @Override
        public void onMetadataUpdate(Cluster cluster) {

            // AUTO_PATTERN 模式的处理
            if (subscriptions.hasPatternSubscription()) {

                ... .. // 权限验证 (略)
                final List<String> topicsToSubscribe = new ArrayList<>();
                for (String topic : cluster.topics())
                    if (filterTopic(topic)) // 通过 subscribedPattern 匹配 Topic
                        topicsToSubscribe.add(topic);

                // 更新 subscriptions 集合、groupSubscription 集合、assignment 集合
                subscriptions.changeSubscription(topicsToSubscribe);
                // 更新 Metadata 需要记录元数据的 Topic 集合
                metadata.setTopics(subscriptions.groupSubscription());
            } else if (!cluster.unauthorizedTopics().isEmpty()) {
                // 抛出异常 (略)
            }

            // 检测是否为 AUTO_PATTERN 或 AUTO_TOPICS 模式
            if (subscriptions.partitionsAutoAssigned()) {
                MetadataSnapshot snapshot = new MetadataSnapshot(
                    subscriptions, cluster); // 创建快照
                if (!snapshot.equals(metadataSnapshot)) { // 比较快照
                    metadataSnapshot = snapshot; // 记录快照
                }
            }
        }
    });
}
```

```

        // 更新 needsPartitionAssignment 字段为 true, 表示需要重新进行分区分配
        subscriptions.needReassignment();
    }
}
});
}

```

- **assignmentSnapshot**: 也是用来存储 Metadata 的快照信息, 不过是用来检测 Partition 分配的过程中有没有发生分区数量变化。具体是在 Leader 消费者开始分区分配操作前, 使用此字段记录 Metadata 快照; 收到 SyncGroupResponse 后, 会比较此字段记录的快照与当前 Metadata 是否发生变化。如果发生变化, 则要重新进行分区分配。在后面的介绍中还会分析上述过程。

3.4.4 PartitionAssignor 分析

Leader 消费者在收到 JoinGroupResponse 后, 会按照其中指定的分区分配策略进行分区分配, 每个分区分配策略就是一个 PartitionAssignor 接口的实现。图 3-17 是 PartitionAssignor 的继承结构及其中的组件。

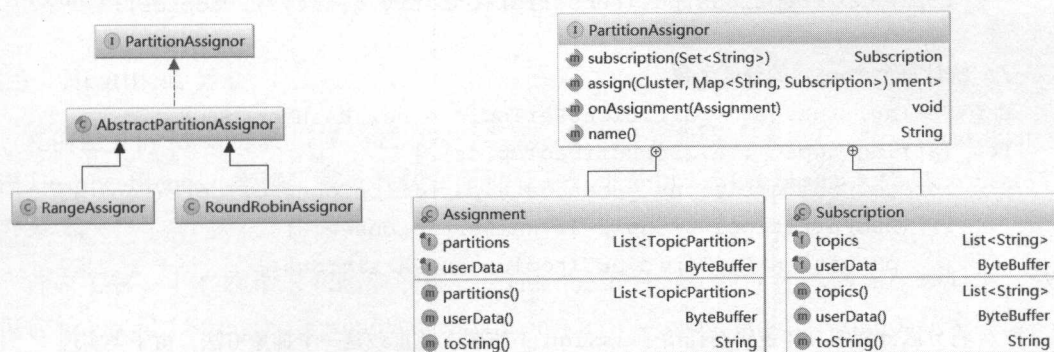


图 3-17

PartitionAssignor 接口中定义了 Assignment 和 Subscription 两个内部类。进行分区分配需要的两方面的数据: Metadata 中记录的集群元数据和每个 Member 的订阅信息。为了用户增强对分配结果的控制, 就将用户订阅信息和一些影响分配的用户自定义信息封装成 Subscription, 例如, “用户自定义数据” 可以是每个消费者的权重。其中, topics 集合表示某 Member 订阅的 Topic 集合, userData 表示用户自定义的数据。PartitionAssignor 接口

提供了 `subscription()` 方法，用于添加用户自定义数据，在创建 `JoinGroupRequest` 的时候会用到 `subscription()` 方法，后面详述。

`Assignment` 中保存了分区的分配结果，`partitions` 表示的是分配给某消费者的 `TopicPartition` 集合，`userData` 是用户自定义的数据。

再看看 `PartitionAssignor` 的其他方法，`assign()` 是子类要实现的、完成 `Partition` 分配的抽象方法。`onAssignment()` 方法是在每个消费者收到 `Leader` 分配结果时的回调函数，此调用发生在解析 `SyncGroupResponse` 之后，后面详述。

`AbstractPartitionAssignor` 为了简化 `PartitionAssignor` 接口的实现，对 `assign()` 方法进行了实现，其中会将 `Subscription` 中的 `userData` 去除掉后，再进行分区分配。具体代码如下：

```
public Map<String, Assignment> assign(Cluster metadata,
    Map<String, Subscription> subscriptions) {
    Set<String> allSubscribedTopics = new HashSet<>();
    Map<String, List<String>> topicSubscriptions = new HashMap<>();
    for (Map.Entry<String, Subscription> subscriptionEntry :
        subscriptions.entrySet()) { // 解析 subscriptions 集合，去除 userData 信息
        List<String> topics = subscriptionEntry.getValue().topics();
        allSubscribedTopics.addAll(topics);
        topicSubscriptions.put(subscriptionEntry.getKey(), topics);
    }
    // 统计每个 Topic 的分区个数
    Map<String, Integer> partitionsPerTopic = new HashMap<>();
    for (String topic : allSubscribedTopics) {
        Integer numPartitions = metadata.partitionCountForTopic(topic);
        if (numPartitions != null && numPartitions > 0)
            partitionsPerTopic.put(topic, numPartitions);
    }
    // 将分区分配的具体逻辑委托给了 assign() 重载，此重载是一个抽象方法，由子类实现
    Map<String, List<TopicPartition>> rawAssignments =
        assign(partitionsPerTopic, topicSubscriptions);

    // 整理分区分配结果
    Map<String, Assignment> assignments = new HashMap<>();
    for (Map.Entry<String, List<TopicPartition>> assignmentEntry :
        rawAssignments.entrySet())
        assignments.put(assignmentEntry.getKey(),
```



```

        new Assignment(assignmentEntry.getValue()));
    return assignments;
}

```

如果开发人员在自定义 PartitionAssignor 时需要使用 userData 控制分区分配结果，就不能直接继承 AbstractPartitionAssignor，而需要直接实现 PartitionAssignor 接口。AbstractPartitionAssignor 中的其他方法比较简单，不再赘述。

RangeAssignor 和 RoundRobinAssignor 都是 Kafka 提供的 PartitionAssignor 接口的默认实现，简单介绍一下，感兴趣的读者可以自行参考代码学习。

- RangeAssignor 实现原理是：针对每个 Topic， $n = \text{分区数} / \text{消费者数量}$ ， $m = \text{分区数} \% \text{消费者数量}$ ，前 m 个消费者每个分配 $n+1$ 个分区，后面的（消费者数量 $-m$ ）个消费者每个分配 n 个 Partition。
- RoundRobinAssignor 原理是：将所有 Topic 的 Partition 按照字典序排列，然后对每个 Consumer 进行轮询分配。

举个例子，有 C0、C1 两个消费者和 t0、t1 两个 Topic，每个 Topic 有三个分区编号都是 0 ~ 2。使用 RangeAssignor 的分配结果是：C0: [t0p0, t0p1, t1p0, t1p1]，C1: [t0p2, t1p2]；使用 RoundRobinAssignor 的分配结果是：C0: [t0p0, t0p2, t1p1]、C1: [t0p1, t1p0, t1p2]。

3.4.5 Heartbeat 分析

在前面分析 Rebalance 操作的原理时介绍到，消费者定期向服务端的 GroupCoordinator 发送 HeartbeatRequest 来确定彼此在线。下面就来详细分析 KafkaConsumer 中 Heartbeat 的相关实现。

首先了解一下心跳请求和响应的格式。HeartbeatRequest 的消息体格式比较简单，依次包含 group_id (String)、group_generation_id (int)、member_id (String) 三个字段。HeartbeatResponse 消息体只包含一个 short 类型的 error_code。

HeartbeatTask 是一个实现 DelayedTask 接口的定时任务，负责定时发送 HeartbeatRequest 并处理其响应，此逻辑在其 run() 方法中实现，下面就来分析 HeartbeatTask.run() 方法的具体流程，如图 3-18 所示。

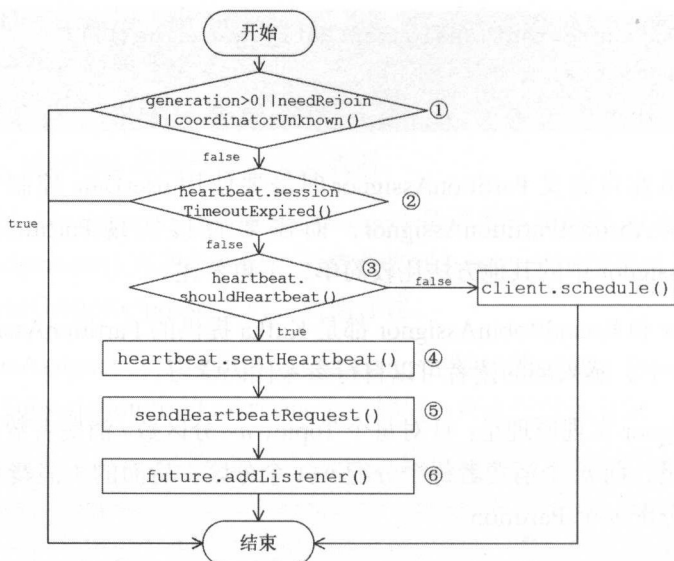


图 3-18

(1) 首先检查是否需要发送 HeartbeatRequest, 条件有多个:

- GroupCoordinator 已确定且已连接;
- 不处于正在等待 Partition 分配结果的状态;
- 之前的 HeartbeatRequest 请求正常收到响应且没有过期。

如果不符合条件, 则不再执行 HeartbeatTask, 等待后续调用 reset() 方法重启 HeartbeatTask 任务。

(2) 调用 heartbeat.sessionTimeoutExpired() 方法, 检测 HeartbeatResponse 是否超时。若超时, 则认为 GroupCoordinator 宕机, 调用 coordinatorDead() 方法清空其 unsent 集合中对应的请求队列并将这些请求标记为异常后结束, 将 coordinator 字段设置为 null, 表示将重新选择 GroupCoordinator。同时还会停止 HeartbeatTask 的执行。coordinatorDead() 方法的代码如下:

```
protected void coordinatorDead() {
    if (this.coordinator != null) {
        // 将 unsent 中缓存的要发送给 Coordinator 节点的请求全部清空，并标记为异常后结束
        client.failUnsentRequests(this.coordinator,
            GroupCoordinatorNotAvailableException.INSTANCE);
        this.coordinator = null; // 将 coordinator 设置为 null
    }
}
```

(3) 检测 HeartbeatTask 是否到期，如果不到期则更新其到期时间，将 HeartbeatTask 对象重新添加到 DelayedTaskQueue 中，等待其到期后执行；如果已到期则继续后面的步骤，发送 HeartbeatRequest 请求。

(4) 更新最近一次发送 HeartbeatRequest 请求的时间，将 requestInFlight 设置为 true，表示有未响应的 HeartbeatRequest 请求，防止重复发送。

(5) 创建 HeartbeatRequest 请求，并调用 ConsumerNetworkClient.send() 方法，将请求放入 unsent 集合中缓存并返回 RequestFuture<Void>。在后面的 ConsumerNetworkClient.poll() 操作中会将其发送给 GroupCoordinator。

(6) 在 RequestFuture<Void> 对象上添加 RequestFutureListener。

HeartbeatTask.run() 方法的具体实现如下：

```
public void run(final long now) {
    if (generation < 0 || needRejoin() || coordinatorUnknown()) {
        return; // 步骤 1: 检查是否要发送心跳请求
    }
    if (heartbeat.sessionTimeoutExpired(now)) {
        coordinatorDead(); // 步骤 2: 上次心跳响应超时，将 GroupCoordinator 标记为宕机
        return;
    }
    if (!heartbeat.shouldHeartbeat(now)) { // 步骤 3: 没有到发送心跳请求的时间。
        client.schedule(this, now + heartbeat.timeToNextHeartbeat(now));
    } else { // 下面是步骤 4
        heartbeat.sentHeartbeat(now); // 更新发送 HeartbeatRequest 的时间
        requestInFlight = true; // 防止重复发送 HeartbeatRequest
        // 步骤 5: 创建并缓存 HeartbeatRequest
        RequestFuture<Void> future = sendHeartbeatRequest();
    }
}
```

```

// 步骤 6: 添加监听器
future.addListener(new RequestFutureListener<Void>() {
    @Override
    public void onSuccess(Void value) {
        requestInFlight = false;
        ... ..// 更新 Heartbeat 记录的时间 (略)
        // 重新调度 HeartbeatTask
        client.schedule(HeartbeatTask.this, nextHeartbeatTime);
    }

    @Override
    public void onFailure(RuntimeException e) {
        requestInFlight = false;
        client.schedule(HeartbeatTask.this, time.milliseconds() +
retryBackoffMs);
    }
});
}
}

```

下面介绍一下 HeartbeatResponse 相关的处理。首先需要注意上面介绍的 sendHeartbeatRequest() 方法，它使用 HeartbeatCompletionHandler 将 client.send() 方法返回的 RequestFuture<ClientResponse> 适配成 RequestFuture<Void> 后返回，代码如下：

```

public RequestFuture<Void> sendHeartbeatRequest() {
    HeartbeatRequest req = new HeartbeatRequest(this.groupId,
        this.generation, this.memberId); // 创建 HeartbeatRequest

    // 使用 HeartbeatCompletionHandler 对 RequestFuture<ClientResponse> 进行适配
    return client.send(coordinator, ApiKeys.HEARTBEAT, req).compose(
        new HeartbeatCompletionHandler());
}

```

在 HeartbeatCompletionHandler 中实现的是 HeartbeatResponse 核心逻辑。下面分析一下 HeartbeatCompletionHandler，图 3-19 展示了其继承关系。

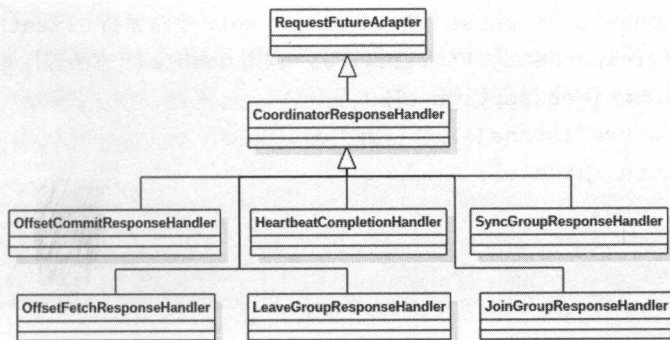


图 3-19

CoordinatorResponseHandler 是一个抽象类，其中有 parse() 和 handle() 两个抽象方法，parse() 方法对 ClientResponse 进行解析，得到指定类型的响应；handle() 方法对解析后的响应进行处理。CoordinatorResponseHandler 实现了 RequestFuture 抽象类的 onSuccess() 方法和 onFailure() 方法。具体代码如下：

```

protected abstract class CoordinatorResponseHandler<R, T> extends
    RequestFutureAdapter<ClientResponse, T> {
    protected ClientResponse response; // 待处理的响应

    // parse() 和 handle() 两个抽象方法
    public abstract R parse(ClientResponse response);
    public abstract void handle(R response, RequestFuture<T> future);

    @Override
    public void onFailure(RuntimeException e, RequestFuture<T> future) {
        // mark the coordinator as dead
        if (e instanceof DisconnectException)
            coordinatorDead();
        future.raise(e); // 调用 adapted 对象的 raise() 方法
    }

    @Override
    public void onSuccess(ClientResponse clientResponse,
        RequestFuture<T> future) {
        try {
            this.response = clientResponse;
        }
    }
  
```



```

    R responseObj = parse(clientResponse); // 解析 ClientResponse
    handle(responseObj, future); // 调用 handle() 方法进行处理, 子类实现
} catch (RuntimeException e) {
    if (!future.isDone())
        future.raise(e);
}
}
}

```

这里使用的是模板方法模式, 由父类模板方法定义操作流程, 由子类根据需求个性化实现流程中的抽象方法。这种设计方式避免了在每个子类中都有一份流程控制代码。

处理 `HeartbeatResponse` 的相关处理流程如图 3-20 所示。

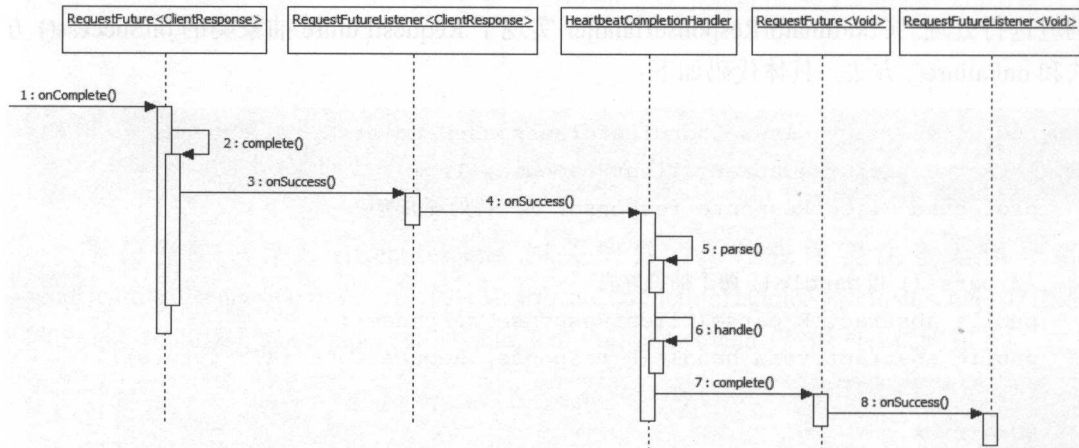


图 3-20

`RequestFuture<ClientResponse>` 和 `RequestFutureListener<ClientResponse>` 只是为了实现适配器的功能, 并没有实际处理逻辑。当 `ClientResponse` 传递到 `HeartbeatCompletionHandler` 处时, 会通过 `parse()` 方法解析成 `HeartbeatResponse`, 然后进入 `handle()` 方法处理。

在 `HeartbeatCompletionHandler.handle()` 方法中, 判断 `HeartbeatResponse` 中是否包含错误码, 如果不包含, 则调用 `RequestFuture<Void>` 的 `complete(null)` 方法, 将 `HeartbeatResponse` 成功的事件传播下去; 反之, 针对不同类型错误码分类处理, 并调用 `raise()` 方法设置对应异常。例如, 错误码是 `ILLEGAL_GENERATION`, 表示 `HeartbeatRequest` 中携带的 `generationId` 过期, `GroupCoordinator` 已经开始新一轮 `Rebalance` 操作, 则将 `rejoinNeeded` 设置为 `true`, 这会重

新发送 JoinGroupRequest 请求尝试加入 Consumer Group，也会导致 HeartbeatTask 任务停止。如果错误码是 UNKNOWN_MEMBER_ID，表示 GroupCoordinator 识别不了此 Consumer，则清空 memberId，尝试重新加入 Consumer Group。剩余错误码的分析不再一一介绍，留给读者自己分析。下面来看其 handle() 方法的具体实现代码：

```
public void handle(HeartbeatResponse heartbeatResponse, RequestFuture<Void>
future) {
    // 解析错误码
    Errors error = Errors.forCode(heartbeatResponse.errorCode());
    if (error == Errors.NONE) { // 心跳正常
        future.complete(null);
    } else if (error == Errors.GROUP_COORDINATOR_NOT_AVAILABLE ||
        // 找不到服务端对应的 GroupCoordinator
        error == Errors.NOT_COORDINATOR_FOR_GROUP) {
        // 清空 unsent 集合中对应的请求，并重新查找对应的 GroupCoordinator
        coordinatorDead();
        future.raise(error); // 设置
    } else if (error == Errors.REBALANCE_IN_PROGRESS) {
        // 重新发送 JoinGroupRequest 消息
        AbstractCoordinator.this.rejoinNeeded = true;
        future.raise(Errors.REBALANCE_IN_PROGRESS);
    } else if (error == Errors.ILLEGAL_GENERATION) {
        // 重新发送 JoinGroupRequest 消息
        AbstractCoordinator.this.rejoinNeeded = true;
        future.raise(Errors.ILLEGAL_GENERATION);
    } else if (error == Errors.UNKNOWN_MEMBER_ID) {
        memberId = JoinGroupRequest.UNKNOWN_MEMBER_ID; // 清空 memberId
        // 重新发送 JoinGroupRequest 消息
        AbstractCoordinator.this.rejoinNeeded = true;
        future.raise(Errors.UNKNOWN_MEMBER_ID);
    } else if (error == Errors.GROUP_AUTHORIZATION_FAILED) {
        future.raise(new GroupAuthorizationException(groupId));
    } else {
        future.raise(new KafkaException("..."));
    }
}
```

HeartbeatCompletionHandler.handle() 方法中会调用 RequestFuture<Void> 的 complete() 方法或

raise() 方法，这两个方法中没有处理逻辑，但是会触发其上的 RequestFutureListener<Void>（在 HeartbeatTask.run() 方法的步骤 6 中注册），此监听器会将 requestInFlight 设置为 false，表示所有 HeartbeatRequest 都已经完成，并将 HeartbeatTask 重新放入定时任务队列，等待下一次到期执行，具体代码不贴出来了。

3.4.6 Rebalance 实现

本节主要介绍 KafkaConsumer 中与 Rebalance 操作相关的实现。在开始介绍 Rebalance 操作的实现细节之前，我们需要明确在哪几种情况下会触发 Rebalance 操作：

（1）有新的消费者加入 Consumer Group。

（2）有消费者宕机下线。消费者并不一定需要真正下线，例如遇到长时间的 GC、网络延迟导致消费者长时间未向 GroupCoordinator 发送 HeartbeatRequest 时，GroupCoordinator 会认为消费者下线。

（3）有消费者主动退出 Consumer Group。

（4）Consumer Group 订阅的任一 Topic 出现分区数量的变化。

（5）消费者调用 unsubscribe() 取消对某 Topic 的订阅。

下面我们开始对 Rebalance 操作的具体实现进行分析。

第一阶段

Rebalance 操作的第一步就是查找 GroupCoordinator，这个阶段消费者会向 Kafka 集群中的任意一个 Broker 发送 GroupCoordinatorRequest 请求，并处理返回的 GroupCoordinatorResponse 响应。

GroupCoordinatorRequest 消息体的格式比较简单，只包含了 Consumer Group 的 id。GroupCoordinatorResponse 消息体包含了错误码（short 类型）、coordinator 的节点 Id（int 类型）、GroupCoordinator 的 host（String 类型）、GroupCoordinator 的端口号（int 类型）。

发送 GroupCoordinatorRequest 请求的入口是 ConsumerCoordinator 的 ensureCoordinatorReady() 方法，其流程如图 3-21 所示。

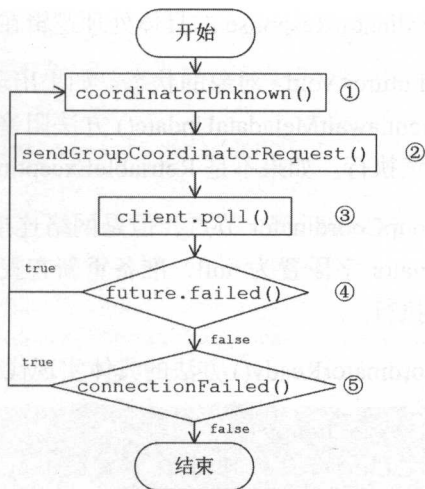


图 3-21

(1) 首先检测是否需要重新查找 GroupCoordinator，主要是检查 coordinator 字段是否为空以及与 GroupCoordinator 之间的连接是否正常。代码如下：

```

public boolean coordinatorUnknown() {
    if (coordinator == null) // 检测 Coordinator 字段是否为 null
        return true;

    // 检测与 GroupCoordinator 之间的网络连接是否正常
    if (client.connectionFailed(coordinator)) {
        // 将 unsent 集中对应的请求清空并将 coordinator 字段设置为 null
        coordinatorDead();
        return true;
    }
    return false;
}

```

(2) 查找集群负载最低的 Node 节点，并创建 GroupCoordinatorRequest 请求。调用 `client.send()` 方法将请求放入 `unsent` 队列中等待发送，并返回 `RequestFuture<Void>` 对象。返回的 `RequestFuture<Void>` 对象经过了 `compose()` 方法适配，原理同 `HeartbeatCompletionHandler`。

(3) 调用 `ConsumerNetworkClient.poll(future)` 方法，将 GroupCoordinatorRequest 请求发送出去。此处使用阻塞的方式发送，直到收到 GroupCoordinatorResponse 响应或异常完成，

才从此方法返回。GroupCoordinatorResponse 的具体处理逻辑在后面介绍。

(4) 检测检查 RequestFuture<Void> 对象的状态。如果出现 RetriableException 异常, 则调用 ConsumerNetworkClient.awaitMetadataUpdate() 方法阻塞更新 Metadata 中记录的集群元数据后跳转到步骤 1 继续执行。如果不是 RetriableException 异常则直接报错。

(5) 如果成功找到 GroupCoordinator 节点, 但是网络连接失败, 则将其 unsent 中对应的请求清空, 并将 coordinator 字段置为 null, 准备重新查找 GroupCoordinator, 退避一段时间后跳转到步骤 1 继续执行。

下面来看一下 ensureCoordinatorReady() 方法的具体实现代码:

```
public void ensureCoordinatorReady() {
    while (coordinatorUnknown()) { // 步骤 1: 检查 GroupCoordinator 的状态
        RequestFuture<Void> future = sendGroupCoordinatorRequest();
        // 步骤 2: 创建并缓存请求
        // 步骤 3: 阻塞发送 GroupCoordinatorRequest, 并处理 GroupCoordinatorResponse
        client.poll(future);
        if (future.failed()) { // 步骤 4: 异常处理
            if (future.isRetriable()) {
                client.awaitMetadataUpdate(); // 阻塞更新 Metadata 中记录的集群元数据
            } else {
                throw future.exception();
            }
        } else if (coordinator != null && client.connectionFailed(coordinator)) {
            coordinatorDead(); // 步骤 5: 连接不到 GroupCoordinator, 退避一段时间, 重试
            time.sleep(retryBackoffMs);
        }
    }
}
```

上述代码中的大部分功能已经在前面分析过了, 这里重点来看一下 sendGroupCoordinatorRequest() 方法的实现:


```

private RequestFuture<Void> sendGroupCoordinatorRequest() {
    // 查找负载最低的节点，底层实现是查找 InFlightRequests 中未确认请求最少的节点，请
    // 读者参考源码
    Node node = this.client.leastLoadedNode();

    if (node == null) { // 找不到可用的节点，则直接返回一个异常结束的 RequestFuture
        return RequestFuture.noBrokersAvailable();
    } else {
        GroupCoordinatorRequest metadataRequest = new GroupCoordinatorRequest(
            this.groupId); // 创建 GroupCoordinatorRequest 请求
        // 将 GroupCoordinatorRequest 缓存到 unsent 集合，ConsumerNetworkClient.send() 方法
        // 在前面介绍过了，不再赘述。这里需要注意的是下面使用的匿名 RequestFutureAdapter
        return client
            .send(node, ApiKeys.GROUP_COORDINATOR, metadataRequest)
            .compose(new RequestFutureAdapter<ClientResponse, Void>() {
                @Override
                public void onSuccess(ClientResponse response,
                    RequestFuture<Void> future)
                {
                    // 处理 GroupMetadataResponse 的入口
                    handleGroupMetadataResponse(response, future);
                }
            });
    }
}

```

下面介绍处理 GroupCoordinatorResponse 的相关操作。通过对 sendGroupCoordinatorRequest() 方法的分析我们知道，handleGroupMetadataResponse() 方法是处理 GroupCoordinatorResponse 的入口，其步骤如下：

- (1) 调用 coordinatorUnknown() 检测是否已经找到 GroupCoordinator 且成功连接。如果是则忽略此 GroupCoordinatorResponse，因为在发送 GroupCoordinatorRequest 时并没有防止重发的机制，可能有多个 GroupCoordinatorResponse；否则，继续下面的步骤。
- (2) 解析 GroupCoordinatorResponse 得到服务端 GroupCoordinator 的信息。
- (3) 构建 Node 对象赋值给 coordinator 字段，并尝试与 GroupCoordinator 建立连接。
- (4) 启动 HeartbeatTask 定时任务。

(5) 最后, 调用 `RequestFuture.complete()` 方法将正常收到 `GroupCoordinatorResponse` 的事件传播出去。

(6) 如果 `GroupCoordinatorResponse` 中的错误码不为 `NONE`, 则调用 `RequestFuture.raise()` 方法将异常传播出去。最终由 `ensureCoordinatorReady()` 方法中的步骤 4 处理。

`handleGroupMetadataResponse()` 方法的具体实现如下:

```
private void handleGroupMetadataResponse(ClientResponse resp,
    RequestFuture<Void> future) {
    if (!coordinatorUnknown()) { // 步骤 1: 检测是否有正常工作的 GroupCoordinator
        future.complete(null);
    } else {
        GroupCoordinatorResponse groupCoordinatorResponse = new
        GroupCoordinatorResponse(
            resp.responseBody()); // 步骤 2: 解析 GroupCoordinatorResponse
        Errors error = Errors.forCode(groupCoordinatorResponse.errorCode());
        if (error == Errors.NONE) {
            // 步骤 3: 创建 GroupCoordinator 对应的 Node 对象
            this.coordinator = new Node(...);
            client.tryConnect(coordinator); // 尝试连接 GroupCoordinator
            if (generation > 0) {
                heartbeatTask.reset(); // 步骤 4: 启动 HeartbeatTask
            }
            future.complete(null); // 步骤 5
        } else if (error == Errors.GROUP_AUTHORIZATION_FAILED) {
            future.raise(new GroupAuthorizationException(groupId));
        } else {
            future.raise(error); // 步骤 6
        }
    }
}
```

第二阶段

在成功查找到对应的 `GroupCoordinator` 之后进入 `Join Group` 阶段。在此阶段, 消费者会向 `GroupCoordinator` 发送 `JoinGroupRequest` 请求, 并处理响应。先来了解 `JoinGroupRequest` 和 `JoinGroupResponse` 的消息体格式, 如图 3-22 所示。

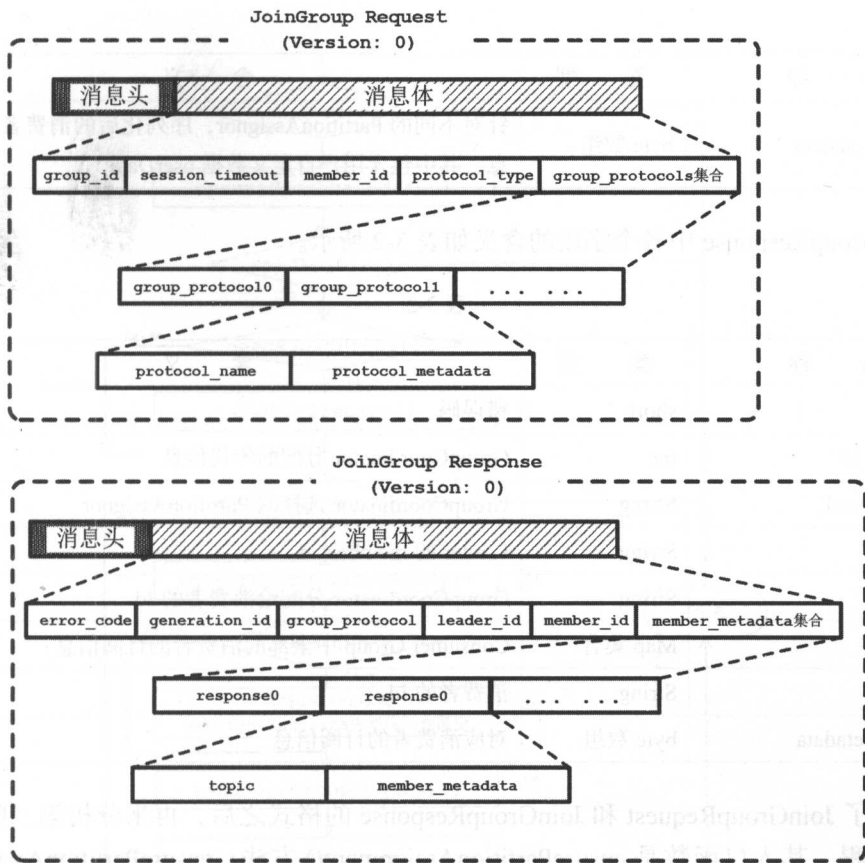


图 3-22

JoinGroupRequest 中各个字段的含义如表 3-1 所示。

表 3-1

名 称	类 型	含 义
group_id	String	Consumer Group 的 Id
session_timeout	int	GroupCoordinator 超过 session_time 指定的时间，没有收到心跳，认为消费者下线
member_id	String	GroupCoordinator 分配给消费者的 Id
protocol_type	String	Consumer Group 实现的协议，默认是“consumer”
group_protocols	List	包含此消费者支持的全部 PartitionAssignor 类型
protocol_name	String	PartitionAssignor 的名称

(续表)

名 称	类 型	含 义
protocol_metadata	byte 数组	针对不同的 PartitionAssignor，序列化后的消费者的订阅信息，其中包含用户自定义数据 userData

JoinGroupResponse 中各个字段的含义如表 3-2 所示。

表 3-2

名 称	类 型	含 义
error_code	short	错误码
generation_id	int	GroupCoordinator 分配的世代信息
group_protocol	String	GroupCoordinator 选择的 PartitionAssignor
leader_id	String	Leader 的 member_id
member_id	String	GroupCoordinator 分配给消费者的 Id
members	Map 集合	Consumer Group 中全部的消费者的订阅信息
member_id	String	消费者的 Id
member_metadata	byte 数组	对应消费者的订阅信息

了解了 JoinGroupRequest 和 JoinGroupResponse 的格式之后，再来分析第二阶段的相关处理流程，其入口函数是 ensurePartitionAssignment() 方法。ensurePartitionAssignment() 方法的流程如图 3-23 所示。

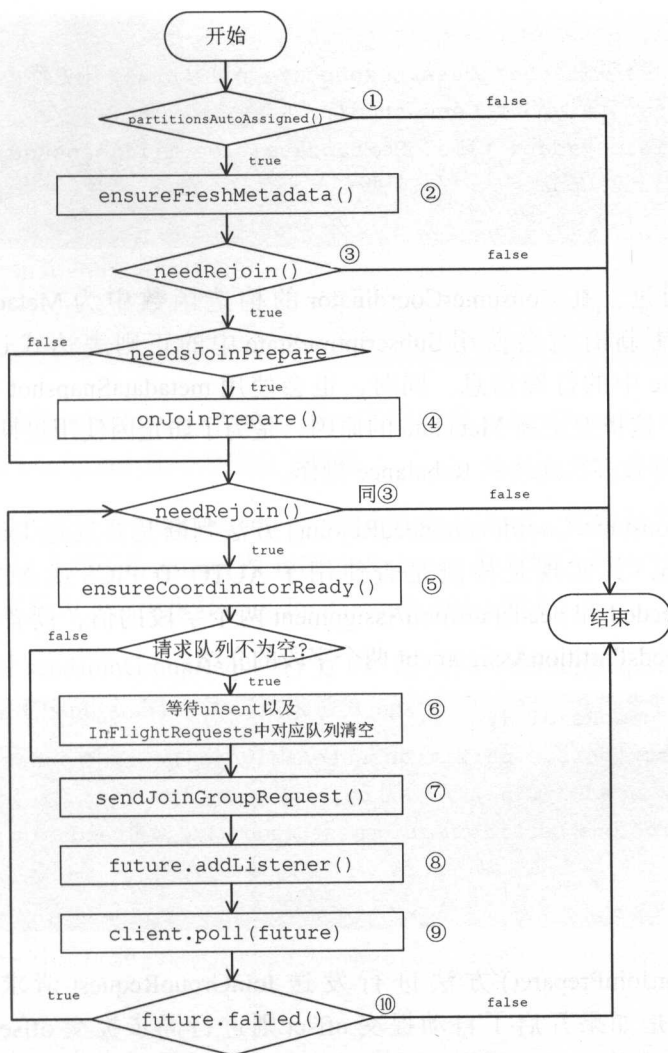


图 3-23

(1) 调用 `SubscriptionState.partitionsAutoAssigned()` 方法，检测 Consumer 的订阅是否是 `AUTO_TOPICS` 或 `AUTO_PATTERN`。因为 `USER_ASSIGNED` 不需要进行 `Rebalance` 操作，而是由用户手动指定分区。

(2) 如果订阅模式是 `AUTO_PATTERN`，则检查 `Metadata` 是否需要更新。


```
public void ensureFreshMetadata() {
    // 如果长时间没有更新或 Metadata.needUpdate 字段为 true, 则更新 Metadata
    if (this.metadata.updateRequested()
        || this.metadata.timeToNextUpdate(time.milliseconds()) == 0)
        awaitMetadataUpdate(); // 阻塞
}
```

在前面提到过, 在 ConsumerCoordinator 的构造函数中为 Metadata 添加了监听器。当 Metadata 更新时就会使用 SubscriptionState 中的正则表达式过滤 Topic, 并更改 SubscriptionState 中的订阅信息。同时, 也会使用 metadataSnapshot 字段记录当前的 Metadata 的快照。这里要更新 Metadata 的原因, 是为了防止因使用过期的 Metadata 进行 Rebalance 操作而导致多次连续的 Rebalance 操作。

(3) 调用 ConsumerCoordinator.needRejoin() 方法判断是要发送 JoinGroupRequest 加入 ConsumerGroup, 其实现是检测是否使用了 AUTO_TOPICS 或 AUTO_PATTERN 模式, 检测 rejoinNeeded 和 needsPartitionAssignment 两个字段的值, 读者可以参考前面对 rejoinNeeded 和 needsPartitionAssignment 两个字段的介绍。

```
public boolean needRejoin() {
    return subscriptions.partitionsAutoAssigned() // 检测 subscriptionType
        && (super.needRejoin() // 检测 rejoinNeeded 的值
        || subscriptions.partitionAssignmentNeeded() // 检测 needsPartitionAssignment
        );
}
```

(4) 调用 onJoinPrepare() 方法进行发送 JoinGroupRequest 请求之前的准备, 做了三件事: 一是如果开启了自动提交 offset 则进行同步提交 offset, 提交 offset 的内容后面会详细介绍, 此步骤可能会阻塞线程; 二是调用注册在 SubscriptionState 中的 ConsumerRebalanceListener 上的回调方法; 三是将 SubscriptionState 的 needsPartitionAssignment 字段设置为 true 并收缩 groupSubscription 集合。

```
protected void onJoinPrepare(int generation, String memberId) {
    maybeAutoCommitOffsetsSync(); // 进行一次同步提交 offset 操作
    // 调用 SubscriptionState 中设置的 ConsumerRebalanceListener
    ConsumerRebalanceListener listener = subscriptions.listener();
    try {
        Set<TopicPartition> revoked = new HashSet<>()
```

```

        subscriptions.assignedPartitions());
        listener.onPartitionsRevoked(revoked);
    } catch (Exception e) {
        // 异常处理(略)
    }

    assignmentSnapshot = null;
    subscriptions.needReassignment();// 将needsPartitionAssignment 设置 true
}

```

(5) 再次调用 `needRejoin()` 方法检测, 之后调用 `ensureCoordinatorReady()` 方法检测已经找到 `GroupCoordinator` 且与之建立了连接。

(6) 如果还有发往 `GroupCoordinator` 所在 `Node` 的请求, 则阻塞等待这些请求全部发送完成并收到响应 (即等待 `unsent` 及 `InFlightRequests` 的对应队列为空), 然后返回步骤 5 继续执行, 主要是为了避免重复发送 `JoinGroupRequest` 请求。

(7) 调用 `sendJoinGroupRequest()` 方法创建 `JoinGroupRequest` 请求, 并调用 `ConsumerNetworkClient.send()` 方法将请求放入 `unsent` 中缓存, 等待发送。具体实现如下:

```

private RequestFuture<ByteBuffer> sendJoinGroupRequest() {
    if (coordinatorUnknown()) // 检测 GroupCoordinator
        return RequestFuture.coordinatorNotAvailable();
    // 创建 JoinGroupRequest
    JoinGroupRequest request = new JoinGroupRequest(groupId,
        this.sessionTimeoutMs, this.memberId, protocolType(), metadata());

    // 将 JoinGroupRequest 放入 unsent 集合等待发送
    // 注意, JoinGroupResponseHandler 是 JoinGroupResponse 处理的入口
    return client.send(coordinator, ApiKeys.JOIN_GROUP, request).compose(
        new JoinGroupResponseHandler());
}

```

(8) 在步骤 7 返回的 `RequestFuture<ByteBuffer>` 对象上添加 `RequestFutureListener`。

(9) 调用 `ConsumerNetworkClient.poll()` 方法发送 `JoinGroupRequest`, 这里会阻塞等待, 直到收到 `JoinGroupResponse` 或出现异常。

(10) 检测 `RequestFuture.fail()`。如果出现 `RetriableException` 异常则进行重试, 其他异常则报错。如果无异常, 则整个第二阶段操作完成。

JoinGroupRequest 的发送流程到这里就已经分析完了。下面来看一下 ensurePartitionAssignment() 方法的代码:

```
public void ensurePartitionAssignment () {
    if (subscriptions.partitionsAutoAssigned()) { // 步骤 1: 检测订阅类型
        if (subscriptions.hasPatternSubscription()) {
            client.ensureFreshMetadata(); // 步骤 2: 检测是否需要更新 Metadata
            ensureActiveGroup(); // 在其中执行后面的步骤
        }
    }
}

public void ensureActiveGroup() {
    if (!needRejoin()) return; // 步骤 3: 检测是否需要发送 JoinGroupRequest 请求
    if (needsJoinPrepare) {
        // 步骤 4: 发送 JoinGroupRequest 请求前的准备操作
        onJoinPrepare(generation, memberId);
        needsJoinPrepare = false;
    }
    while (needRejoin()) {
        ensureCoordinatorReady(); // 步骤 5: 检测 GroupCoordinator 状态
        if (client.pendingRequestCount(this.coordinator) > 0) {
            // 步骤 6: 等待发往 GroupCoordinator 所在节点的消息全部完成
            client.awaitPendingRequests(this.coordinator);
            continue;
        }
        // 步骤 7: 创建并缓存请求
        RequestFuture<ByteBuffer> future = sendJoinGroupRequest();
        // 步骤 8: 添加监听器
        future.addListener(new RequestFutureListener<ByteBuffer>() {
            ... // 此 RequestFutureListener 的代码属于处理 JoinGroupResponse 部分, 后面会详述
        });
        client.poll(future); // 步骤 9: 阻塞等待 JoinGroupRequest 请求完成
        if (future.failed()) { // 步骤 10: 异常处理
            ...
            time.sleep(retryBackoffMs); // 退避一段时间重试。
        }
    }
}
```

通过前面对 JoinGroupRequest 发送流程的分析,我们了解到 JoinGrouResponse 处理流程的入口是 JoinGroupResponseHandler.handle() 方法,其中还包括了 SyncGroupRequest 发送的操作,后面详述。JoinGrouResponse 的处理流程如图 3-24 所示。

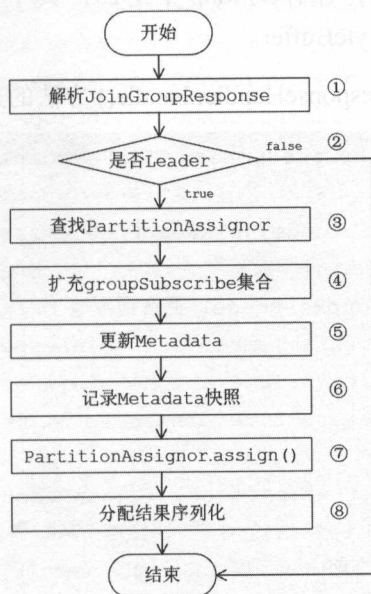


图 3-24

(1) 解析 JoinGroupResponse, 获取 GroupCoordinator 分配的 memberId、generation 等信息, 更新到本地。

(2) 消费者根据 leaderId 检测自己是不是 Leader。如果是 Leader 则进入 onJoinLeader() 方法, 如果不是 Leader 则进入 onJoinFollower() 方法。从上面的流程图也可以看出, onJoinFollower() 方法的逻辑是 onJoinLeader() 方法的子集, 下面主要分析 onJoinLeader() 方法。

(3) Leader 根据 JoinGroupResponse 的 group_protocol 字段指定的 Partition 分配策略, 查找相应的 PartitionAssignor 对象。

(4) Leader 将 JoinGroupResponse 的 members 字段进行反序列化, 得到 Consumer Group 中全部消费者订阅的 Topic。Leader 会将这些 Topic 信息添加到其 SubscriptionState.groupSubscription 集合中。而 Follower 则只关心自己订阅的 Topic 信息。

(5) 第 4 步可能有新的 Topic 添加进来, 所以更新 Metadata 信息。

(6) 待 Metadata 更新完成后, 会在 assignmentSnapshot 字段中存储一个 Metadata 快

照（即通过 Metadata 的 Listener 创建的快照）。

（7）调用 `PartitionAssignor.assign()` 方法进行分区分配。

（8）将分配结果序列化，保存到 Map 中返回，其中 key 是消费者的 `memberId`，value 是分配结果序列化后的 `ByteBuffer`。

下面来看一下 `JoinGroupResponseHandler.handle()` 方法的这段处理的代码：

```
public void handle(JoinGroupResponse joinResponse, RequestFuture<ByteBuffer> future) {
    Errors error = Errors.forCode(joinResponse.errorCode());
    if (error == Errors.NONE) {
        // 步骤 1: 解析 JoinGroupResponse, 更新到本地
        AbstractCoordinator.this.memberId = joinResponse.memberId();
        AbstractCoordinator.this.generation = joinResponse.generationId();
        AbstractCoordinator.this.rejoinNeeded = false; // 注意, 修改 rejoinNeeded 这个标记
        AbstractCoordinator.this.protocol = joinResponse.groupProtocol();
        if (joinResponse.isLeader()) { // 步骤 2: 判断是否为 Leader
            // 注意这里, 此 future 是在前面 sendJoinGroupRequest() 方法返回的 RequestFuture 对象
            // 在 onJoinLeader() 和 onJoinFollower() 方法中, 都涉及发送 SyncGroupRequest
            // 逻辑, 返回的 RequestFuture 标识的是 SyncGroupRequest 的完成情况。这里使用
            // chain() 方法, 主要实现的功能是: 当 SyncGroupResponse 处理完成后, 再通知此
            // future 对象。了解这个逻辑, 可以帮着读者理清处理流程
            onJoinLeader(joinResponse).chain(future);
        } else {
            onJoinFollower().chain(future);
        }
    } else {
        future.raise(new KafkaException(...)); // 异常处理 (略)
    }
}

// 下面是 onJoinLeader() 方法的实现
private RequestFuture<ByteBuffer> onJoinLeader(JoinGroupResponse joinResponse, int i) {
    try {
        // 步骤 3~8 都是在 performAssignment() 方法中完成的
```



```

Map<String, ByteBuffer> groupAssignment = performAssignment(
    joinResponse.leaderId(), joinResponse.groupProtocol(),
    joinResponse.members());
Map<String, ByteBuffer> groupAssignment = new HashMap<>();
// 创建并发送 SyncGroupRequest
SyncGroupRequest request = new SyncGroupRequest(groupId,
    generation, memberId, groupAssignment);
return sendSyncGroupRequest(request);
} catch (RuntimeException e) {
    return RequestFuture.failure(e);
}
}

// 下面是 performAssignment() 方法的实现
protected Map<String, ByteBuffer> performAssignment(String leaderId,
    String assignmentStrategy, Map<String, ByteBuffer> allSubscriptions)
{
    // 步骤 3: 查找分区分配使用的 PartitionAssignor
    PartitionAssignor assignor = lookupAssignor(assignmentStrategy);
    ... .. // 反序列化操作及汇总操作 (略)
    // 步骤 4: 对于 Leader 来说, 要关注 Consumer Group 中全部消费者订阅的 Topic
    this.subscriptions.groupSubscribe(allSubscribedTopics);
    metadata.setTopics(this.subscriptions.groupSubscription());
    client.ensureFreshMetadata(); // 步骤 5: 更新 Metadata
    assignmentSnapshot = metadataSnapshot; // 步骤 6: 记录快照
    ... .. // 日志输出 (略)
    // 步骤 7: 进行分区分配
    Map<String, Assignment> assignment = assignor.assign(metadata.fetch(),
        subscriptions);
    // 步骤 8: 将分区分配结果序列化, 并保存到 groupAssignment 中
    Map<String, ByteBuffer> groupAssignment = new HashMap<>();
    for (Map.Entry<String, Assignment> assignmentEntry : assignment.
        entrySet()) {
        ... .. // 序列化操作 (略)
        groupAssignment.put(assignmentEntry.getKey(), buffer);
    }
    return groupAssignment;
}

```

第三阶段

完成分区分配之后就进入了 Synchronizing Group State 阶段，主要逻辑是向 GroupCoordinator 发送 SyncGroupRequest 请求并处理 SyncGroupResponse 响应。先来了解 SyncGroupRequest 和 SyncGroupResponse 的消息体格式，如图 3-25 所示。

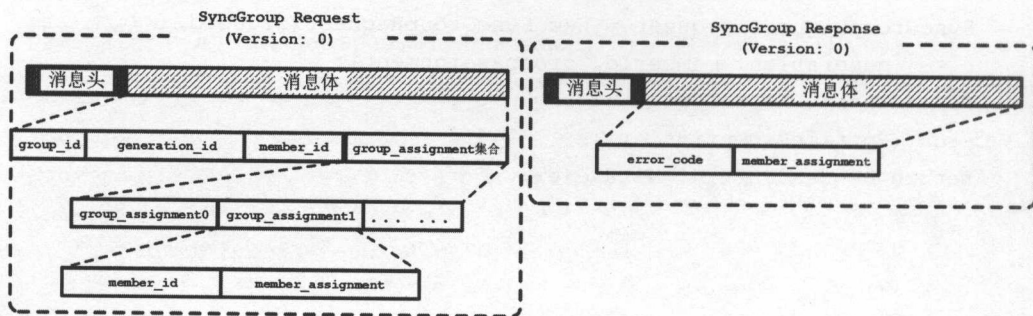


图 3-25

SyncGroupRequest 中各个字段的含义如表 3-3 所示。

表 3-3

名 称	类 型	含 义
group_id	String	Consumer Group 的 Id
generation_id	int	消费者保存的年代信息
member_id	String	GroupCoordinator 分配给消费者的 Id
member_assignment	byte 数组	分区分配结果

SyncGroupResponse 中各个字段的含义如表 3-4 所示。

表 3-4

名 称	类 型	含 义
error_code	short	错误码
member_assignment	byte 数组	分配给当前消费者的分区

通过前面对 onJoinLeader() 方法分析，我们知道发送 SyncGroupRequest 请求的逻辑紧接在分区分配操作之后，也是在 onJoinLeader() 方法中完成的。下面是其流程：

(1) 得到序列化后的分区分配结果后，Leader 将其封装成 SyncGroupRequest，而

Follower 形成的 SyncGroupRequest 中这部分为空集合。

(2) 调用 ConsumerNetworkClient.send() 方法将请求放入 unsent 集合中等待发送。

对 SyncGroupResponse 处理的入口是 SyncGroupResponseHandler.handle() 方法。对于正常完成的情况, 解析 SyncGroupResponse, 从中拿到分区分配结果并将其传递出去; 对于出现异常情况, 将 rejoinNeeded 设置为 true, 并针对不同的错误码进行不同的处理。

```
public void handle(SyncGroupResponse syncResponse, RequestFuture<ByteBuffer>
future) {
    Errors error = Errors.forCode(syncResponse.errorCode());
    if (error == Errors.NONE) {
        // 调用 RequestFuture.complete() 方法传播分区分配结果
        future.complete(syncResponse.memberAssignment());
    } else {
        // 将 rejoinNeeded 设置为 true
        AbstractCoordinator.this.rejoinNeeded = true;
        if (error == Errors.GROUP_AUTHORIZATION_FAILED) {
            // 调用 RequestFuture.raise() 方法传播异常
            future.raise(new GroupAuthorizationException(groupId));
        }
        ... ..// 其他异常的处理方式类似
    }
}
```

从 SyncGroupResponse 中得到的分区分配结果最终由 ConsumerCoordinator.onJoinComplete() 方法处理, 调用此方法的是在第二阶段 ensureActiveGroup() 方法的步骤 8 中添加的 RequestFutureListener 中调用。onJoinComplete() 方法的流程如图 3-26 所示。

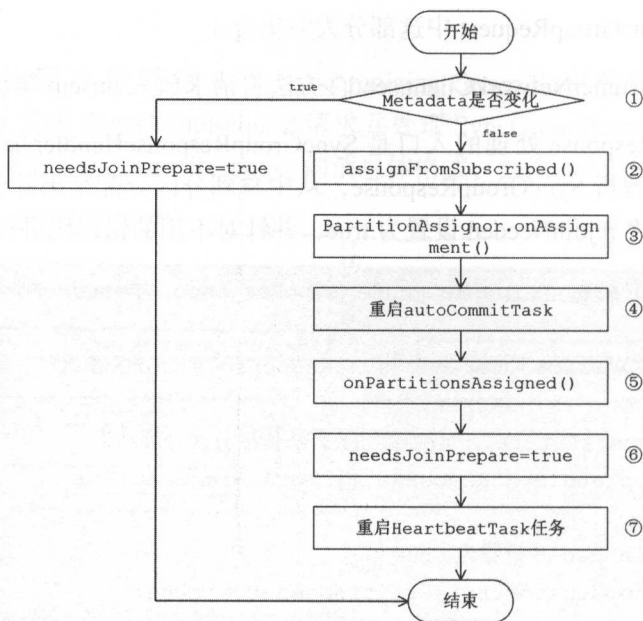


图 3-26

(1) 在第二阶段 Leader 开始分配分区之前, Leader 使用 `assignmentSnapshot` 字段记录了 Metadata 快照。此时在 Leader 中, 将此快照与最新的 Metadata 快照进行对比。如果快照不一致则表示分区分配过程中出现了 Topic 增删或分区数量的变化, 则将 `needsPartitionAssignment` 置为 `true`, 需重新进行分区分配。

(2) 反序列化拿到分配给当前消费者的分区, 并添加到 `SubscriptionState.assignment` 集合中, 之后消费者会按照此集合指定的分区进行消费, 将 `needsPartitionAssignment` 置为 `false`。

(3) 调用 `PartitionAssignor` 的 `onAssignment()` 回调函数, 默认是空实现。当用户自定义 `PartitionAssignor` 时, 可以自定义此方法。

(4) 如果开启了自动提交 offset 的功能, 则重新启动 `AutoCommitTask` 定时任务。

(5) 调用 `SubscriptionState` 中注册的 `ConsumerRebalanceListener`。

(6) 将 `needsJoinPrepare` 重置为 `true`, 为下次 Rebalance 操作做准备。

(7) 重启 `HeartbeatTask` 定时任务, 定时发送心跳。

下面 `onJoinComplete()` 方法的具体实现, 其中省略了异常处理和边界检查的代码:


```

protected void onJoinComplete(int generation, String memberId,
    String assignmentStrategy, ByteBuffer assignmentBuffer) {
    if (assignmentSnapshot != null// 步骤1: Leader 需要比较快照, 而 Follower 则不用
        && !assignmentSnapshot.equals(metadataSnapshot)) {
        subscriptions.needReassignment();
        return;
    }
    // 查找使用的分配策略
    PartitionAssignor assignor = lookupAssignor(assignmentStrategy);
    // 步骤2: 反序列化, 更新 assignment
    Assignment assignment = ConsumerProtocol.deserializeAssignment(assignmentBuffer);

    // 将 needsFetchCommittedOffsets 设置为 true, 允许从服务端获取最近一次提交的 offset
    subscriptions.needRefreshCommits();
    // 填充 assignment 集合
    subscriptions.assignFromSubscribed(assignment.partitions());
    assignor.onAssignment(assignment); // 步骤3: 回调函数
    if (autoCommitEnabled) // 步骤4: 开启 AutoCommitTask 定时任务
        autoCommitTask.reschedule();
    ConsumerRebalanceListener listener = subscriptions.listener();
    // 步骤5: 回调 ConsumerRebalanceListener
    listener.onPartitionsAssigned(assigned);
    needsJoinPrepare = true; // 步骤6: 修改 needsJoinPrepare
    heartbeatTask.reset();// 步骤7: 开启 HeartbeatTask
}

```

到此为止, Rebalance 操作的执行流程和具体实现就分析完了。当 Consumer 正常离开 ConsumerGroup 时会发送 LeaveGroupRequest, 此时也会触发 Rebalance 操作, 逻辑比较简单, 请读者参考源码学习。

3.4.7 offset 操作

提交 offset

通过前面的分析, 我们了解 Rebalance 操作的原理和实现。在进行消费者正常消费过程中以及 Rebalance 操作开始之前, 都会提交一次 offset 记录 Consumer 当前的消费位置。提交 offset 的功能也是由 ConsumerCoordinator 实现的。

先来了解 OffsetCommitRequest 和 OffsetCommitResponse 的消息体格式，如图 3-27 所示。

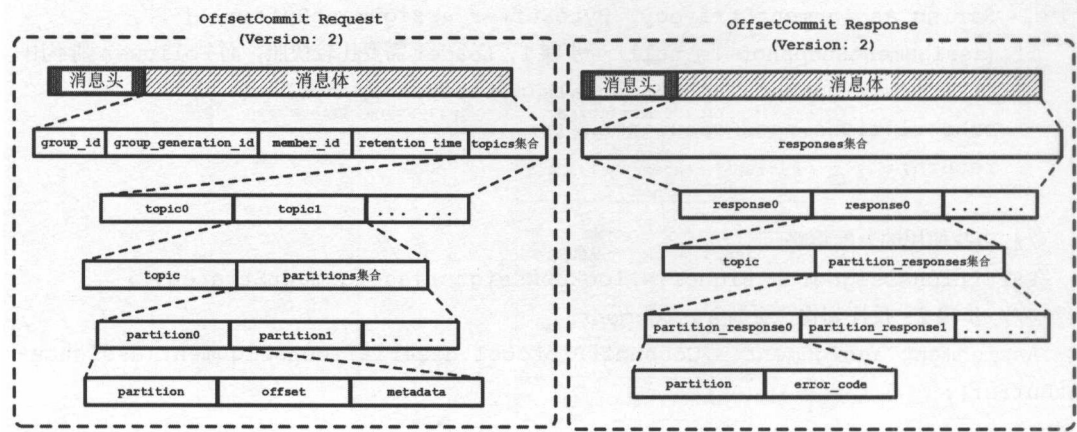


图 3-27

OffsetCommitRequest 中各个字段的含义如表 3-5 所示。

表 3-5

名 称	类 型	含 义
group_id	String	Consumer Group 的 Id
group_generation_id	int	消费者保存的年代信息
member_id	String	GroupCoordinator 分配给消费者的 Id
retention_time	long	此 offset 的最长保存时间
topic	String	topic 名称
partition	int	分区编号
offset	long	提交的消息 offset
metadata	String	任何希望与 offset 一起保存的自定义数据

OffsetCommitResponse 中各个字段的含义如表 3-6 所示。

表 3-6

名 称	类 型	含 义
topic	String	topic 名称
partition	int	分区编号
error_code	short	错误码

图 3-28 展示了 ConsumerCoordinator 中与提交 offset 相关的四个方法以及它们之间的调用关系。

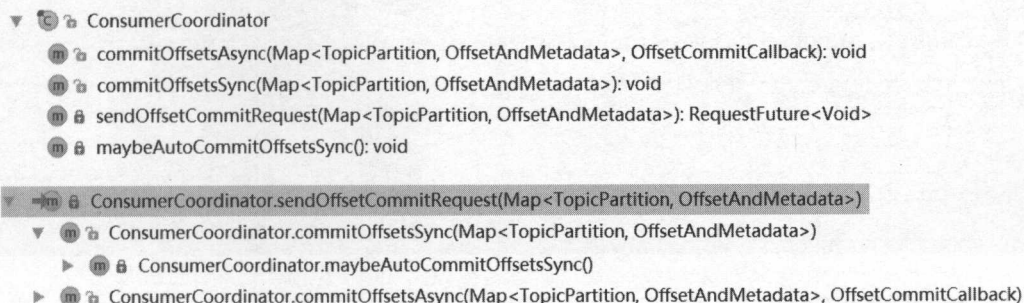


图 3-28

在前面介绍过，在 SubscriptionState 中使用 TopicPartitionState 记录了每个 TopicPartition 的消费状况，TopicPartitionState.position 字段则记录了消费者下次要从服务端获取的消息的 offset。当没有明确指定待提交的 offset 值时，则将 TopicPartitionState.position 作为待提交 offset，组织成集合，形成 ConsumerCoordinator.commitOffset*() 方法的第一个参数。读者可以参考 KafkaConsumer.commitOffsetsAsync() 方法以及 AutoCommitTask。下面先来看一下 commitOffsetsAsync() 方法的实现：

```

public void commitOffsetsAsync(final Map<TopicPartition, OffsetAndMetadata>
offsets,
    OffsetCommitCallback callback) {
    this.subscriptions.needRefreshCommits(); // 将 needsFetchCommittedOffsets
    设置为 true

    // 创建并缓存 OffsetCommitRequest 请求，逻辑与之前发送 JoinGroupRequest 和
    // SyncGroupRequest 类似
    // 唯一的区别就是使用 OffsetCommitResponseHandler 处理 OffsetCommitResponse
    RequestFuture<Void> future = sendOffsetCommitRequest(offsets);
    final OffsetCommitCallback cb = callback == null ? defaultOffsetCommitCallback
        : callback; // 选择回调函数
    future.addListener(new RequestFutureListener<Void>() {
        public void onSuccess(Void value) {
            if (interceptors != null) {
                interceptors.onCommit(offsets);
            }
        }
    });
}

```

```

        cb.onComplete(offsets, null); // 调用回调
    }

    public void onFailure(RuntimeException e) {
        ... .. // 异常处理 (略)
    }
});
client.pollNoWakeup(); // 将 OffsetCommitRequest 发送出去
}

```

`commitOffsetsSync()` 方法与 `commitOffsetsAsync()` 方法的实现类似，也是调用 `sendOffsetCommitRequest()` 方法创建并缓存 `OffsetCommitRequest`，使用 `OffsetCommitResponseHandler` 处理 `OffsetCommitResponse`。但是有两点不同：一是 `commitOffsetsSync()` 方法在发送 `OffsetCommitRequest` 时使用了 `ConsumerCoordinator.poll(future)` 阻塞等待 `OffsetCommitResponse` 处理完成，这样才实现了同步提交的功能；二是 `commitOffsetsSync()` 方法在检测到 `RetriableException` 异常时会进行重试。`commitOffsetsSync()` 方法的具体代码就不贴出来了。`maybeAutoCommitOffsetsSync()` 方法会根据 `enable.auto.commit` 配置项的值决定是否调用 `commitOffsetsAsync()` 方法，代码比较简单，不再贴出来了。

`AutoCommitTask` 是一个定时任务，它周期性地调用 `commitOffsetsAsync()` 方法，实现了自动提交 offset 的功能。开启自动提交 offset 功能后，业务逻辑中就可以不用手动调用 `commitOffsets*()` 方法提交 offset 了。`AutoCommitTask` 的代码比较简单，这里就不贴出来了。

`OffsetCommitResponseHandler.handle()` 方法是处理 `OffsetCommitResponse` 的入口，其代码如下：

```

public void handle(OffsetCommitResponse commitResponse, RequestFuture<Void>
future) {
    // 一些初始化操作 (略)
    for (Map.Entry<TopicPartition, Short> entry : commitResponse
        .responseData().entrySet()) { // 遍历待提交的所有 offset 信息
        TopicPartition tp = entry.getKey();
        OffsetAndMetadata offsetAndMetadata = this.offsets.get(tp);
        long offset = offsetAndMetadata.offset();
        Errors error = Errors.forCode(entry.getValue()); // 获取错误码
    }
}

```

```

if (error == Errors.NONE) {
    if (subscriptions.isAssigned(tp))
        // 更新 SubscriptionState 中对应 TopicPartitionState 的 committed 字段
        subscriptions.committed(tp, offsetAndMetadata);
    } else {
        // 出现异常情况时的处理 (略)
    }
}
// 传播成功提交 offset 的事件, 最终调用上面添加的 Callback
future.complete(null);
}

```

fetch offset

在 Rebalance 操作结束之后, 每个消费者都确定了其需要消费的分区。在开始消费之前, 消费者需要确定拉取消息的起始位置。假设之前已经将最后的消费位置提交到了 GroupCoordinator, GroupCoordinator 将其保存到了 Kafka 内部的 Offsets Topic 中, 此时消费者可以通过 OffsetFetchRequest 请求获取上次提交 offset 并从此处继续消费。

OffsetFetchRequest 和 OffsetFetchResponse 的消息体格式如图 3-29 所示。

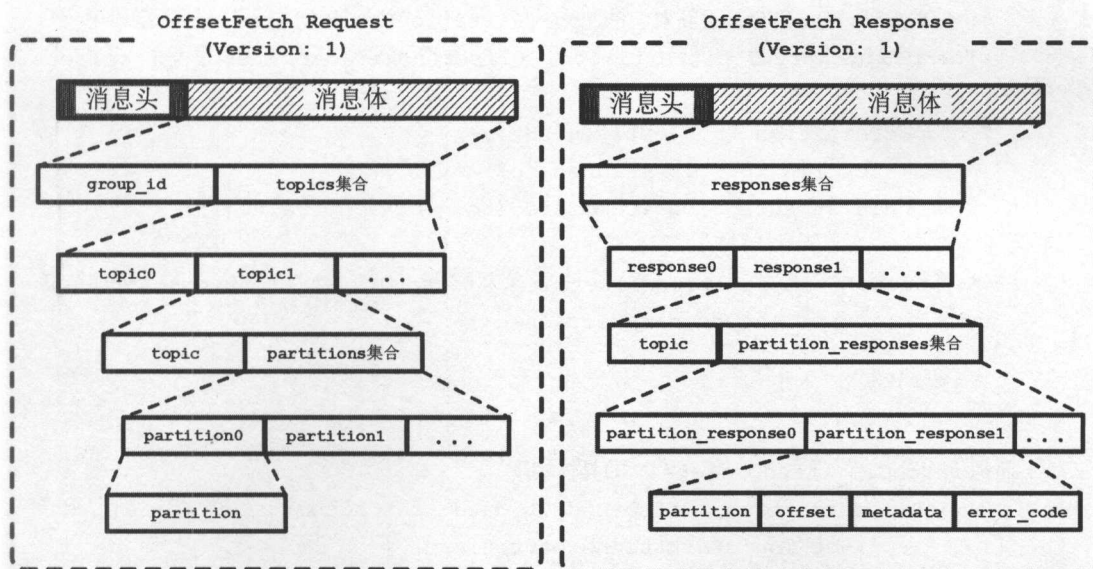


图 3-29

图 3-30 展示了 ConsumerCoordinator 中与 fetch offset 相关的方法及其调用关系。

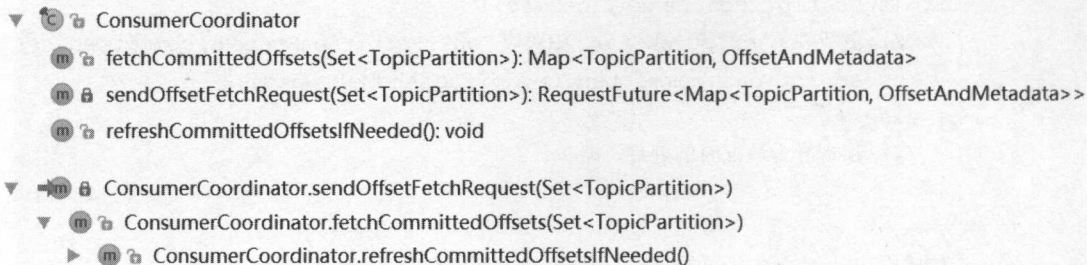


图 3-30

refreshCommittedOffsetsIfNeeded() 方法的主要功能是发送 OffsetFetchRequest 请求，从服务端拉取最近提交的 offset 集合，并更新到 Subscriptions 集合中。下面是其具体实现：

```

public void refreshCommittedOffsetsIfNeeded() {
    if (subscriptions.refreshCommitsNeeded()) { // 检查 needsFetchCommittedOffsets
        // 发送 OffsetFetchRequest 并处理 OffsetFetchResponse 响应。返回值是最近提交
        // 的 offset 集合
        Map<TopicPartition, OffsetAndMetadata> offsets =
            fetchCommittedOffsets(subscriptions.assignedPartitions());
        // 处理 offsets 集合，更新对应的 TopicPartitionState 的 committed 字段中
        for (Map.Entry<TopicPartition, OffsetAndMetadata> entry : offsets.
            entrySet()) {
            TopicPartition tp = entry.getKey();
            if (subscriptions.isAssigned(tp))
                this.subscriptions.committed(tp, entry.getValue());
        }
        // 将 needsFetchCommittedOffsets 设置为 false, OffsetFetchRequest 处理结束...
        this.subscriptions.commitsRefreshed();
    }
}

// 下面是 fetchCommittedOffsets() 的具体实现
public Map<TopicPartition, OffsetAndMetadata> fetchCommittedOffsets(
    Set<TopicPartition> partitions) {
    while (true) {
        ensureCoordinatorReady(); // 检测 GroupCoordinator 的状态
    }
}

```



```

// 创建并缓存 OffsetFetchRequest 请求
RequestFuture<Map<TopicPartition, OffsetAndMetadata>> future =
    sendOffsetFetchRequest(partitions);

client.poll(future); // 阻塞发送 OffsetFetchRequest 请求
if (future.succeeded()) {
    return future.value(); // 返回从服务端得到的 offset
}
// 如果是 RetriableException 异常, 则退避一段时间, 重试
if (!future.isRetriable())
    throw future.exception();
time.sleep(retryBackoffMs);
}
}

```

处理 OffsetFetchResponse 的入口是 OffsetFetchResponseHandler.handle() 方法, 实现如下:

```

public void handle(OffsetFetchResponse response, RequestFuture<Map<TopicParti-
rtition,
    OffsetAndMetadata>> future) {
    Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>()
        response.responseData().size()); // 记录从服务端获取的 offset 的集合
    for (Map.Entry<TopicPartition, OffsetFetchResponse.PartitionData> entry
: response
        .responseData().entrySet()) { // 处理 OffsetFetchResponse 中的每个分区
        TopicPartition tp = entry.getKey();
        OffsetFetchResponse.PartitionData data = entry.getValue();
        if (data.hasError()) {
            // 异常处理过程 (略)
            return;
        } else if (data.offset >= 0) {
            // 记录正常的 offset
            offsets.put(tp, new OffsetAndMetadata(data.offset, data.
metadata));
        }
    }
    // 传播 offsets 集合, 最终通过 fetchCommittedOffsets() 方法返回
    future.complete(offsets);
}
}

```

3.4.8 Fetcher

通过上一节的介绍，我们了解了 offset 操作的原理。本节来了解一下消费者如何从服务端获取消息，KafkaConsumer 依赖 Fetcher 类实现此功能。Fetcher 类的主要功能是发送 FetchRequest 请求，获取指定的消息集合，处理 FetchResponse，更新消费位置。图 3-31 是 Fetcher 类依赖的组件。

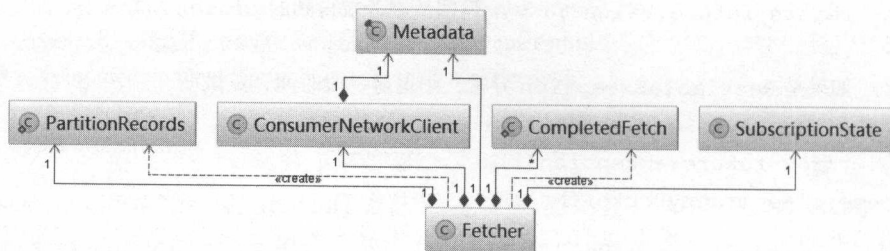


图 3-31

先来了解 Fetcher 的核心字段。

- **client**: `ConsumerNetworkClient`，负责网络通信。
- **minBytes**: 在服务端收到 `FetchRequest` 之后并不是立即响应，而是当可返回的消息数据积累到至少 `minBytes` 个字节时才进行响应。这样每个 `FetchResponse` 中就包含多条消息，提高网络的有效负载。
- **maxWaitMs**: 等待 `FetchResponse` 的最长时间，服务端根据此时间决定何时进行响应。
- **fetchSize**: 每次 `fetch` 操作的最大字节数。
- **maxPollRecords**: 每次获取 `Record` 的最大数量。
- **metadata**: 记录了 `Kafka` 集群的元数据。
- **subscriptions**: 记录每个 `TopicPartition` 的消费情况。
- **completedFetches**: `List<CompletedFetch>` 类型，每个 `FetchResponse` 首先会转换成 `CompletedFetch` 对象进入此队列缓存，此时并未解析消息。
- **keyDeserializer**、**valueDeserializer**: `key` 和 `value` 的反序列化器。
- **nextInLineRecords**: `PartitionRecords` 类型。`PartitionRecords` 保存了 `CompletedFetch` 解析后的结果集合，其中有三个字段：`records` 是消息集合，`fetchOffset` 记录了

records 中第一个消息的 offset, partition 记录了消息对应的 TopicPartition。

Fetcher 的核心方法可以分为三类: fetch 消息的相关方法, 用于从 Kafka 获取消息; 更新 offset 相关的方法, 用于更新 TopicPartitionState 中的 position 字段; 获取 Metadata 信息的方法, 用于获取指定 Topic 的元信息。

Fetch 消息

首先来了解 FetchRequest 和 FetchResponse 的消息体的格式, 如图 3-32 所示。

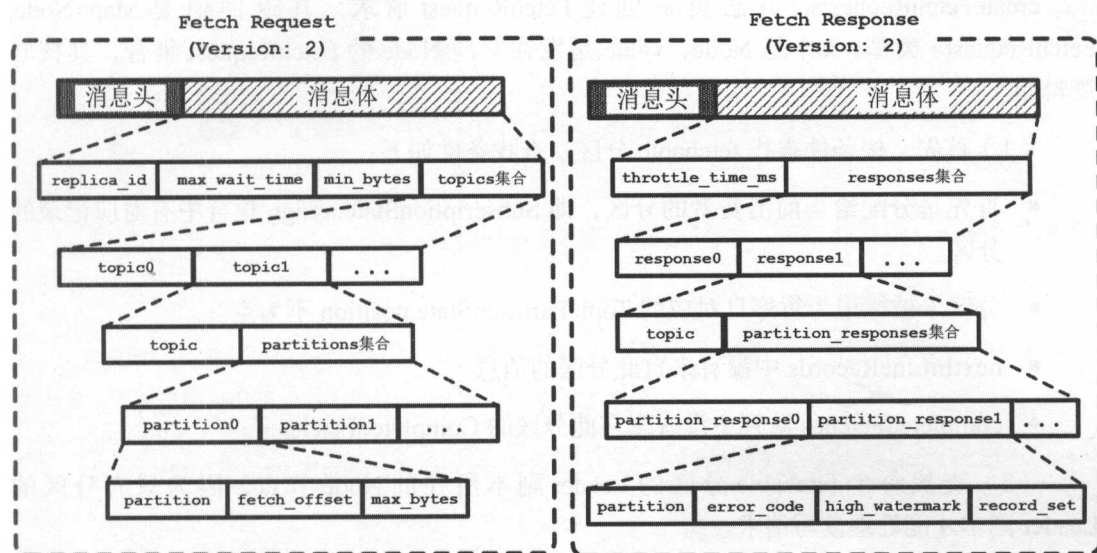


图 3-32

FetchRequest 中的字段如表 3-7 所示。

表 3-7

名 称	类 型	含 义
replica_id	int	用来标识 Follower 的 id, Consumer 和 Follower 都会使用 FetchRequest 从 Leader 那里拉取消息, Consumer 默认是 -1
max_wait_time	int	请求最大的等待时间
min_bytes	int	响应的最小负载
fetch_offset	long	需要 fetch 的消息 offset
max_bytes	int	每次 fetch 的最大字节数

FetchResponse 中的字段如表 3-8 所示。

表 3-8

名 称	类 型	含 义
high_watermark	long	Leader 的 high_watermark
record_set	byte 数组	fetch 到的消息数据

createFetchRequest() 方法负责创建 FetchRequest 请求，其返回值是 Map<Node, FetchRequest> 类型，key 是 Node，value 是发往对应 Node 的 FetchRequest 集合，其核心逻辑如下：

(1) 首先，按条件查找 fetchable 分区。查找条件如下：

- 首先是分配给当前消费者的分区，即 SubscriptionState.assign 集合中有对应记录的分区。
- 分区未被标记为暂停且对应的 TopicPartitionState.position 不为空。
- nextInLineRecords 中没有来自此分区的消息。
- completedFetches 队列中没有来自此分区的 CompletedFetch。

(2) 查找每个 fetchable 分区的 Leader 副本所在的 Node 节点，因为只有分区的 Leader 副本才能处理读写请求。

(3) 检查步骤 2 中找到的 Node 节点，如果其在 unsent 集合或 InFlightRequest 中的对应请求队列不为空，则不对此 Node 发送 FetchRequest 请求。

(4) 通过 SubscriptionState 查找每个分区对应的 position，并封装成 PartitionData 对象。

(5) 最后，按照 Node 进行分类，将发往同一 Node 节点的所有 TopicPartition 封装成一个 FetchRequest 对象。

createFetchRequest() 方法的代码如下：

```
private Map<Node, FetchRequest> createFetchRequest() {
    Cluster cluster = metadata.fetch();    // 获取 Kafka 集群的元数据
    Map<Node, Map<TopicPartition, FetchRequest.PartitionData>> fetchable =
        new HashMap<>();

    // fetchablePartitions() 方法就是按照上述步骤 1 的条件过滤，得到可以发送
    // FetchRequest 的分区
```



```

for (TopicPartition partition : fetchablePartitions()) {
    // 步骤 2: 查找分区的 Leader 副本所在的 Node
    Node node = cluster.leaderFor(partition);
    if (node == null) {
        metadata.requestUpdate(); // 找不到 Leader 副本则准备更新 Metadata
    } // 步骤 3: 是否还有 pending 请求
    if (this.client.pendingRequestCount(node) == 0) {
        Map<TopicPartition, FetchRequest.PartitionData> fetch = fetchable.
get(node);
        ... ..// 初始化操作 (略)
        long position = this.subscriptions.position(partition);
        // 步骤 4: 记录每个分区的对应的 position, 即要 fetch 的消息的 offset
        fetch.put(partition, new FetchRequest.PartitionData(position,
            this.fetchSize));
        // 注意, FetchRequest.PartitionData 不要与 FetchResponse.PartitionData 搞混了
    }
}
// 步骤 5: 对上面的 fetchable 集合进行转换, 将发往同一 Node 节点的所有 TopicPartition
// 的 position 信息封装成一个 FetchRequest 对象
Map<Node, FetchRequest> requests = new HashMap<>();
for (Map.Entry<Node, Map<TopicPartition, FetchRequest.PartitionData>>
entry :
    fetchable.entrySet()) {
    Node node = entry.getKey();
    FetchRequest fetch = new FetchRequest(this.maxWaitMs,
        this.minBytes, entry.getValue());
    requests.put(node, fetch);
}
return requests;
}

```

sendFetches() 方法的主要功能是将 FetchRequest 添加到 unsent 集合中等待发送, 并注册 FetchResponse 处理函数。FetchResponse 的处理主要是解析 FetchResponse 后按照 TopicPartition 分类, 将获取到的消息数据 (未解析的 byte 数组) 和 offset 组装成 CompletedFetch 对象并添加到 completedFetches。sendFetches() 方法的代码如下:


```

public void sendFetches() {
    for (Map.Entry<Node, FetchRequest> fetchEntry : createFetchRequest().
        entrySet()) {
        final FetchRequest request = fetchEntry.getValue();
        // 将发往每个 Node 的 FetchRequest 都缓存到 unsent 队列上
        client.send(fetchEntry.getKey(), ApiKeys.FETCH, request)
            // 添加 Listener, 这也是处理 FetchResponse 的入口
            .addListener(new RequestFutureListener<ClientResponse>() {
                public void onSuccess(ClientResponse resp) {
                    FetchResponse response = new FetchResponse(resp.
                        responseBody());

                    for (Map.Entry<TopicPartition, FetchResponse.
                        PartitionData> entry :
                        // 遍历响应中的数据
                        response.responseData().entrySet()) {
                        TopicPartition partition = entry.getKey();
                        long fetchOffset = request.fetchData().get(partition).
                            offset;

                        // 注意, 这里是 FetchResponse.PartitionData 类型
                        FetchResponse.PartitionData fetchData = entry.
                            getValue();

                        // 创建 CompletedFetch, 并缓存到 completedFetches 队列中
                        completedFetches.add(new CompletedFetch(
                            partition, fetchOffset, fetchData, ...));
                    }
                }
            });

        public void onFailure(RuntimeException e) { // 仅记录错误日志
        }
    }
}

```

存储在 `completedFetches` 队列中的消息数据还是未解析的 `FetchResponse.PartitionData` 对象。在 `fetchedRecords()` 方法中会将 `CompletedFetch` 中的消息数据进行解析, 得到 `Record` 集合并返回, 同时还会修改对应 `TopicPartitionState` 的 `position`, 为下次 `fetch` 操作做好准备。 `fetchedRecords()` 方法的代码如下:

```

public Map<TopicPartition, List<ConsumerRecord<K, V>>> fetchedRecords() {
    if (this.subscriptions.partitionAssignmentNeeded()) {
        return Collections.emptyMap(); // 需要进行 Rebalance 操作则返回空集合
    } else {
        // 按照 TopicPartition 分类
        Map<TopicPartition, List<ConsumerRecord<K, V>>> drained = new
HashMap<>();
        // 一次最多取出 maxPollRecords 条消息
        int recordsRemaining = maxPollRecords;
        Iterator<CompletedFetch> completedFetchesIterator = completedFetches
            .iterator(); // completedFetches 集合的迭代器

        while (recordsRemaining > 0) { // 遍历 completedFetches 集合
            if (nextInLineRecords == null || nextInLineRecords.isEmpty()) {
                if (!completedFetchesIterator.hasNext())
                    break;
                CompletedFetch completion = completedFetchesIterator.
next();

                completedFetchesIterator.remove();
                // 解析 CompletedFetch, 得到一个 PartitionRecords 对象
                nextInLineRecords = parseFetchedData(completion);
            } else {
                recordsRemaining -= append(drained, nextInLineRecords,
                    // 将 nextInLineRecords 中的消息添加到 drained 中
                    recordsRemaining);
            }
        }
        return drained; // 将结果集合返回
    }
}

// parseFetchedData 中实现解析 CompleteFetch, 下面代码将日志输出和异常处理部分省略
private PartitionRecords<K, V> parseFetchedData(CompletedFetch completedFetch) {
    TopicPartition tp = completedFetch.partition;
    FetchResponse.PartitionData partition = completedFetch.partitionData;
    long fetchOffset = completedFetch.fetchedOffset;
    int bytes = 0;
    int recordsCount = 0;
    PartitionRecords<K, V> parsedRecords = null;

```

```

if (partition.errorCode == Errors.NONE.code()) { // 其他错误码的处理省略
    Long position = subscriptions.position(tp);
    ByteBuffer buffer = partition.recordSet;
    // 创建 MemoryRecords, 其中的 ByteBuffer 来自 FetchResponse
    MemoryRecords records = MemoryRecords.readableRecords(buffer);
    List<ConsumerRecord<K, V>> parsed = new ArrayList<>();
    boolean skippedRecords = false;
    // 遍历 MemoryRecords 获取 Record 集合, MemoryRecords 中的迭代器在第 4 章详细分析
    for (LogEntry logEntry : records) {
        // 跳过早于 position 的消息
        if (logEntry.offset() >= position) {
            parsed.add(parseRecord(tp, logEntry));
            bytes += logEntry.size();
        } else {
            skippedRecords = true;
        }
    }
    // 将解析后的 Record 集合封装成 PartitionRecords
    parsedRecords = new PartitionRecords<>(fetchOffset, tp, parsed);
}
return parsedRecords;
}

// 在 fetchedRecords() 方法中将消息添加到 drained 集合中时, 还更新 TopicPartitionState
// 的 position 字段。下面是 Fetcher.append() 方法的代码, 其中省略了一些检查判断和异常
// 处理代码
private int append(Map<TopicPartition, List<ConsumerRecord<K, V>>> drained,
    PartitionRecords<K, V> partitionRecords, int maxRecords)
{
    long position = subscriptions.position(partitionRecords.partition);
    if (partitionRecords.fetchOffset == position) {
        List<ConsumerRecord<K, V>> partRecords = partitionRecords
            .take(maxRecords); // 获取消息集合, 最多 maxRecords 个消息
        long nextOffset = partRecords.get(partRecords.size() - 1)
            .offset() + 1; // 最后一个消息的 offset
        List<ConsumerRecord<K, V>> records = drained.get(partitionRecords.
partition);
        if (records == null) {

```



```

        records = partRecords;
        drained.put(partitionRecords.partition, records);
    } else {
        records.addAll(partRecords);
    }
    // 更新 SubscriptionState 中对应 TopicPartitionState 的 position 字段
    subscriptions.position(partitionRecords.partition, nextOffset);
    return partRecords.size();
}
return 0;
}

```

读者可能注意到，在 `parseFetchedData()` 方法中使用了 `MemoryRecords` 迭代器遍历其中的消息，因为涉及压缩消息的处理，在第4章对 `Log SubSystem` 分析时再具体分析。

更新 position

在有些场景下，例如第一次消费某个 Topic 的分区，服务端的内部 `Offsets Topic` 中并没有记录当前消费者在此分区上的消费位置，所以消费者无法从服务端获取最近提交的 `offset`。此时如果用户手动指定消费的起始 `offset`，则可以从指定 `offset` 开始消费，否则就需要重置 `TopicPartitionState.position` 字段。

重置 `TopicPartitionState.position` 字段的过程中涉及 `OffsetsRequest` 和 `OffsetsResponse`，先来介绍其格式，如图 3-33 所示。在 `OffsetsRequest` 中需要说明的字段是 `timestamp`，取值为 -1 或 -2，分别表示 `LATEST`、`EARLIEST` 两种重置策略。在 `OffsetsResponse` 中需要说明的字段是 `offsets`，它是服务端返回的 `offset` 集合。

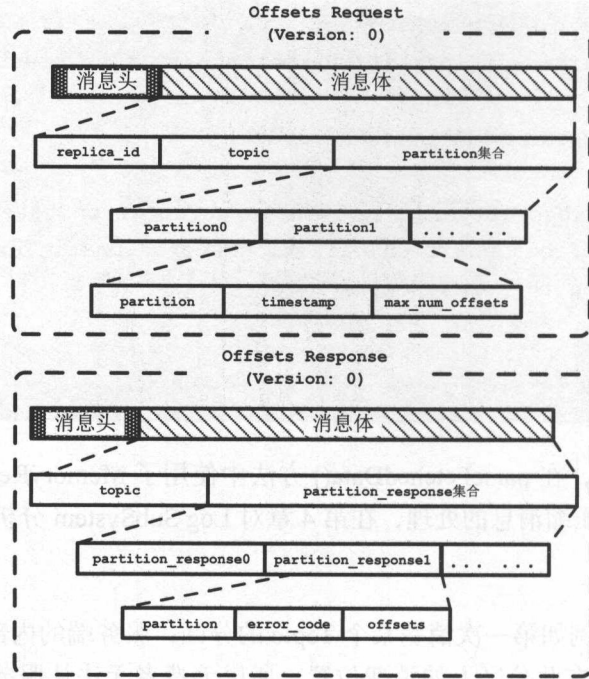


图 3-33

Fetcher.updateFetchPositions() 方法中实现了重置 TopicPartitionState.position 字段的功能，其具体逻辑如下：

- (1) 检测 position 是否为空，如果非空则不需要重置操作。
- (2) 如果设置了 resetStrategy，则按照指定的重置策略进行重置操作。
- (3) 有 EARLIEST、LATEST 两种策略：EARLIEST 策略是将 position 重置为当前最小的 offset；而 LATEST 则是将 position 重置为当前最大的 offset。
- (4) 上面的两种策略都会向 GroupCoordinator 发送 OffsetsRequest，请求指定的 offset。OffsetsRequest 的发送逻辑和 OffsetsResponse 的处理逻辑与前面介绍的类似。
- (5) 如果没有指定重置策略，则将 position 重置为 committed。
- (6) 如果 committed 为空，则使用默认的重置策略。默认重置策略是 LATEST 策略。

Fetcher.updateFetchPositions() 方法具体代码如下：


```

public void updateFetchPositions(Set<TopicPartition> partitions) {
    for (TopicPartition tp : partitions) {
        if (!subscriptions.isAssigned(tp) || subscriptions.isFetchable(tp))
            continue; // 检测 position 是否为空, 如果非空则表示正常, 不需要重置操作

        if (subscriptions.isOffsetResetNeeded(tp)) {
            resetOffset(tp); // 按照指定的策略对 position 进行更新
        } else if (subscriptions.committed(tp) == null) {
            // 对应的 TopicPartitionState.committed 字段为空
            // 则按照 default strategy 策略更新 position 的值
            subscriptions.needOffsetReset(tp);
            resetOffset(tp);
        } else {
            // 如果对应的 TopicPartitionState.committed 字段不为空, 则将 position
            // 字段更新为 committed 字段值
            long committed = subscriptions.committed(tp).offset();
            subscriptions.seek(tp, committed);
        }
    }
}

private void resetOffset(TopicPartition partition) {
    OffsetResetStrategy strategy = subscriptions.resetStrategy(partition);
    final long timestamp; // 根据配置的重置策略选择 timestamp
    if (strategy == OffsetResetStrategy.EARLIEST)
        timestamp = ListOffsetRequest.EARLIEST_TIMESTAMP; // timestamp = -2
    else if (strategy == OffsetResetStrategy.LATEST)
        timestamp = ListOffsetRequest.LATEST_TIMESTAMP; // timestamp = -1
    else
        throw new NoOffsetForPartitionException(partition);

    // 向分区的 Leader 副本所在的 Node 发送一个 OffsetsRequest 请求, 并阻塞等待其响应
    // 处理 OffsetsResponse 后返回 offset
    long offset = listOffset(partition, timestamp);

    if (subscriptions.isAssigned(partition)) {
        this.subscriptions.seek(partition, offset); // 更新 position
    }
}

```

`listOffset()` 方法中实现了对 `OffsetsRequest` 的发送和 `OffsetsResponse` 的处理，与前面介绍的其他请求类似，这里不再赘述，感兴趣的读者可以参考源码进行学习。

获取集群元数据

在 `Fetcher` 中还提供了获取 `Metadata` 信息的相关方法。涉及 `sendMetadataRequest()`、`getTopicMetadata()`、`getAllTopicMetadata()` 三个方法，其调用关系如图 3-34 所示。

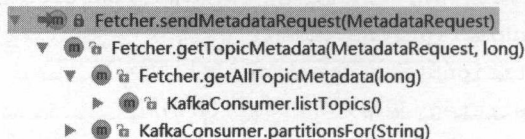


图 3-34

基本逻辑是发送 `MetadataRequest` 请求到负载最小的 `Node` 节点，并阻塞等待 `MetadataResponse`，正常收到响应后对其解析，得到需要的集群元数据。如果读者了解了前面的其他请求的发送及响应处理流程，会发现此处的实现大同小异，不再赘述。

需要注意的是，`Fetcher` 提供的这三个获取集群元数据的相关方法并不会更新 `Fetcher.metadata` 字段中保存的集群元数据。在第 2 章我们介绍过，更新 `Metadata` 使用的是 `NetworkClient.DefaultMetadataUpdater`，同样也是发送的 `MetadataRequest` 请求。`MetadataRequest` 和 `MetadataResponse` 的格式介绍参考第 2 章。

3.4.9 KafkaConsumer 分析总结

到这里为止，我们已经分析完了 `KafkaConsumer` 依赖的相关组件。现在回头来看 `KafkaConsumer` 的整体架构，如图 3-35 所示。`KafkaConsumer` 依赖 `SubscriptionState` 管理订阅的 `Topic` 集合和 `Partition` 的消费状态，通过 `ConsumerCoordinator` 与服务端的 `GroupCoordinator` 交互，完成 `Rebalance` 操作并请求最近提交的 `offset`。`Fetcher` 负责从 `Kafka` 中拉取消息并进行解析，同时参与 `position` 的重置操作，提供获取指定 `Topic` 的集群元数据的操作。上述操作的所有请求都是通过 `ConsumerNetworkClient` 缓存并发送的，在 `ConsumerNetworkClient` 中还维护了定时任务队列，用来完成 `HearbeatTask` 任务和 `AutoCommitTask` 任务。`NetworkClient` 在接收到上述请求的响应时会调用相应回调，最终交给其对应的 `*Handler` 以及 `RequestFuture` 的监听器进行处理。

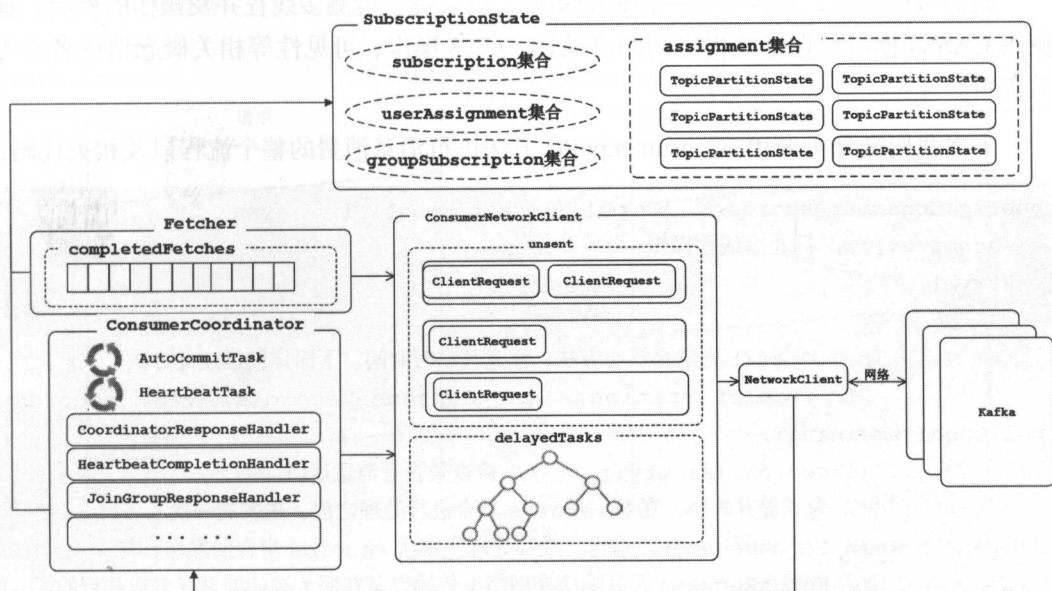


图 3-35

回顾完 `KafkaConsumer` 的整体架构，下面开始分析 `KafkaConsumer` 中剩余的代码。在前面提到过，`KafkaConsumer` 并不是一个线程安全的类。为了防止多线程并发操作，`KafkaConsumer` 提供了多线程并发的检测机制，涉及的方法是 `acquire()` 和 `release()`。这两个方法的代码如下：

```
private void acquire() {
    long threadId = Thread.currentThread().getId();
    // 记录当前线程 Id, 通过 CAS 操作完成
    if (threadId != currentThread.get())
        && !currentThread.compareAndSet(NO_CURRENT_THREAD, threadId)
        // 检测到多线程并发操作, 则报错
        throw new ConcurrentModificationException("...");
    refcount.incrementAndGet(); // 记录重入次数
}

private void release() {
    if (refcount.decrementAndGet() == 0)
        currentThread.set(NO_CURRENT_THREAD); // 更新线程 Id
}
```


我们可以看出，这并不是锁的实现，仅实现了检测多线程并发操作的检测。这里使用 CAS 操作可以保证线程之间的可见性。CAS 操作、可见性等相关概念请读者参考资料。

下面我们来分析 `KafkaConsumer.poll()` 方法进行消息消费的整个流程以及相关代码：

```
public ConsumerRecords<K, V> poll(long timeout) {
    acquire();// 防止多线程操作
    try {
        do {
            // pollOnce() 方法是核心方法，注意其超时时间。下面详细描述此方法
            Map<TopicPartition, List<ConsumerRecord<K, V>>> records =
pollOnce(remaining);
            if (!records.isEmpty()) { // 检查是否有消息返回
                // 为了提升效率，在对 records 集合进行处理之前，先发送一次
                // FetchRequest。这样，线程处理完本次 records 集合的操作，与
                // FetchRequest 及其响应在网络上传输以及在服务端的处理就变成并行的了
                // 这样就可以减少等待网络 I/O 的时间，如图 3-36 所示
                fetcher.sendFetches();// 创建并缓存 FetchRequest

                // 调用 ConsumerNetworkClient.pollNoWakeup() 方法将 FetchRequest
                // 发送出去。注意，此处的 pollNoWakeup() 方法并不会阻塞，不能被中断，
                // 不会执行定时任务
                client.pollNoWakeup();

                // 调用 ConsumerInterceptors
                return this.interceptors.onConsume(new ConsumerRecords<>(records));
            }
            long elapsed = time.milliseconds() - start; // 计算超时时间
            remaining = timeout - elapsed;
        } while (remaining > 0);
        return ConsumerRecords.empty();
    } finally {
        release();
    }
}
```

注意，在消费完消息之后，客户端还需要 commit offset，手动同步 commit offset 使用 `commitSync()`，手动异步 commit offset 使用 `commitAsync()`，自动 commit offset 使用定时

任务 AutoCommitTask。

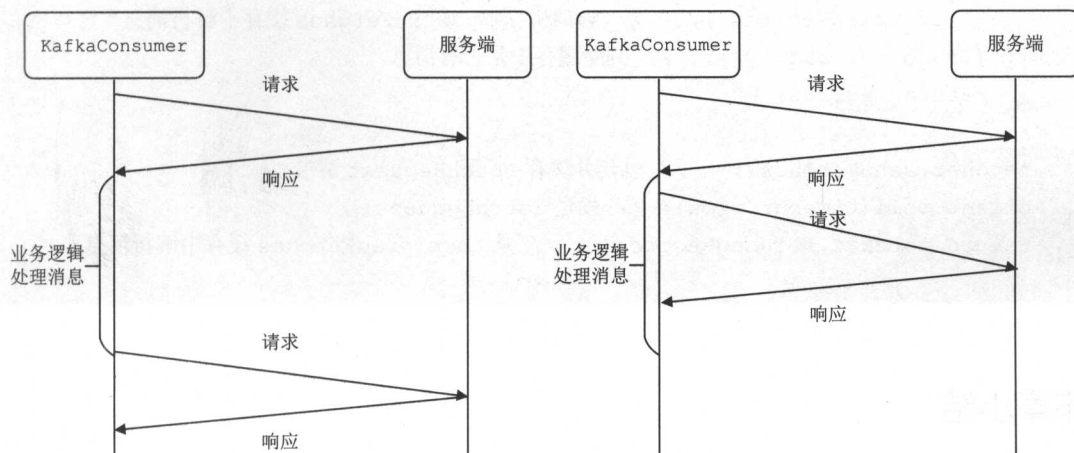


图 3-36

在 `pollOnce()` 方法中会先通过 `ConsumerCoordinator` 与 `GroupCoordinator` 交互完成 `Rebalance` 操作，之后从 `GroupCoordinator` 获取最近一次提交的 `offset`（或重置 `position`），最后才是使用 `Fetcher`，从 `Kafka` 获取消息进行消费。`pollOnce()` 方法的代码如下：

```

private Map<TopicPartition, List<ConsumerRecord<K, V>>> pollOnce(long
timeout) {
// 通过 GroupCoordinatorRequest 查找 GroupCoordinator。如果没找到，会一直阻塞在这里
coordinator.ensureCoordinatorReady();

// AUTO_TOPICS 或 AUTO_PATTERN 订阅模式
if (subscriptions.partitionsAutoAssigned()) {
    coordinator.ensurePartitionAssignment(); // 完成 Rebalance 操作
}

// 恢复 SubscriptionState 中对应的 TopicPartitionState 状态
// 主要是 committed 字段和 position 字段
if (!subscriptions.hasAllFetchPositions()) {
    updateFetchPositions(this.subscriptions.missingFetchPositions());
}

long now = time.milliseconds();
client.executeDelayedTasks(now); // 执行定时任务，HeartbeatTask 和 AutoCommitTask
  
```



```
Map<TopicPartition, List<ConsumerRecord<K, V>>> records = fetcher
    .fetchedRecords(); // 尝试从 completedFetches 缓存中解析消息
if (!records.isEmpty()) // 判断缓存中是否有消息
    return records;

fetcher.sendFetches(); // 创建并缓存 FetchRequest 请求
client.poll(timeout, now); // 发送 FetchRequest
return fetcher.fetchedRecords(); // 从 completedFetches 缓存中解析消息
}
```

本章小结

首先，本章通过一个消费者示例程序向读者介绍了如何通过 `KafkaConsumer` 的 API 实现 `Kafka` 消费者。然后，介绍了消息的传递保证并给出了相关的实践指导，还介绍了 `Consumer Group Rebalance` 各个版本方案的原理和弊端。之后，详细剖析了 `KafkaConsumer` 的相关组件的运行原理和实现细节，介绍了 `ConsumerNetworkClient` 如何在 `NetworkClient` 之上实现更高层的功能，介绍了 `SubscriptionState` 如何管理消费者在每个分区上的消费状态，分析了 `ConsumerCoordinator` 如何与 `GroupCoordinator` 交互、如何实现 `Rebalance` 操作、如何实现提交 `offset` 等相关操作，分析了 `Fetcher` 如何从服务器拉取消息、更新消费位置等功能，还剖析了 `HeartbeatTask` 和 `AutoCommitTask` 定时任务的原理。最后，回顾了 `KafkaConsumer` 的整体架构，通过 `KafkaConsumer.poll()` 和 `pollOnce()` 方法将所有组件串联起来。通过本章的分析，希望读者可以理解消费者的设计原理和实现细节，在实践中更好地应用 `KafkaConsumer`。

第 4 章

Kafka 服务端

在开始介绍 Kafka 服务端的代码之前，先从整体上对其架构了解一下。Kafka Server 的架构如图 4-1 所示。本章会详细介绍每个组件的功能和实现。

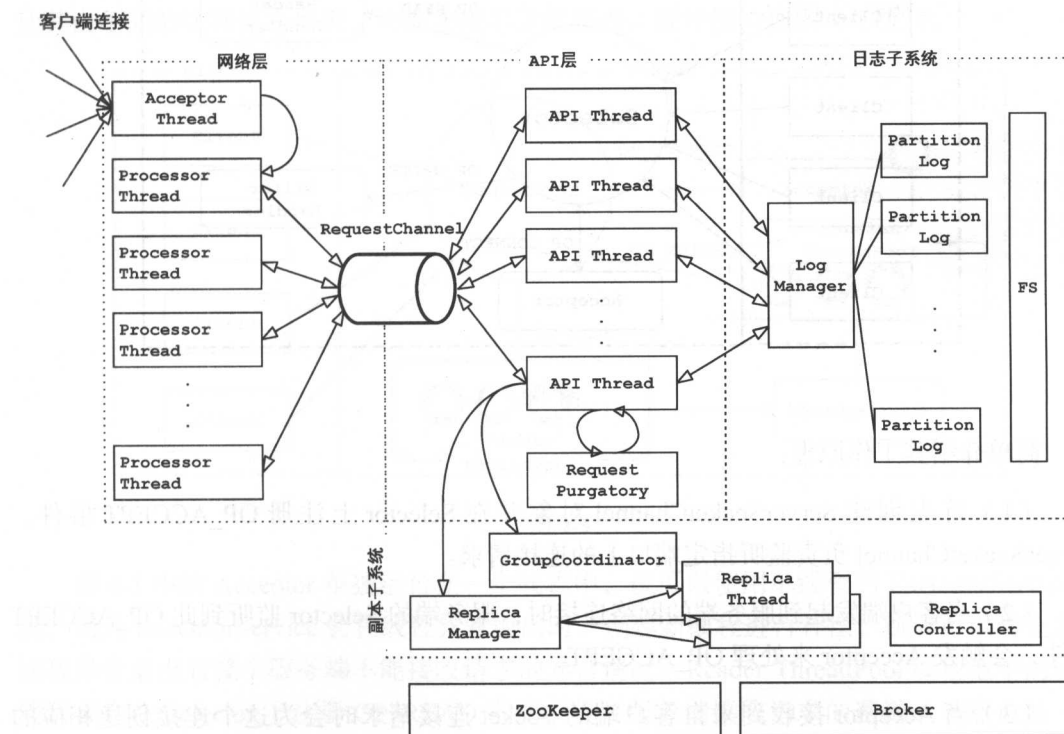


图 4-1

4.1 网络层

通过前面的介绍我们已经了解到，Kafka 的客户端会与服务端的多个 Broker 创建网络连接，在这些网络连接上流转着各种请求及其响应，从而实现客户端与服务端之间的交互。客户端一般情况下不会碰到大数据量访问、高并发的场景，所以客户端使用 `NetworkClient` 组件管理这些网络连接足矣。Kafka 服务端与客户端的运行场景不同，面对高并发、低延迟的需求，Kafka 服务端使用 `Reactor` 模式实现其网络层。Kafka 的网络层管理的网络连接中不仅有来自客户端的，还会有来自其他 Broker 的网络连接。

4.1.1 Reactor 模式

Kafka 网络层采用的是 `Reactor` 模式，是一种基于事件驱动的模式。熟悉 Java 编程的读者应该了解 Java NIO 提供了实现 `Reactor` 模式的 API。常见的单线程 Java NIO 的编程模式如图 4-2 所示。

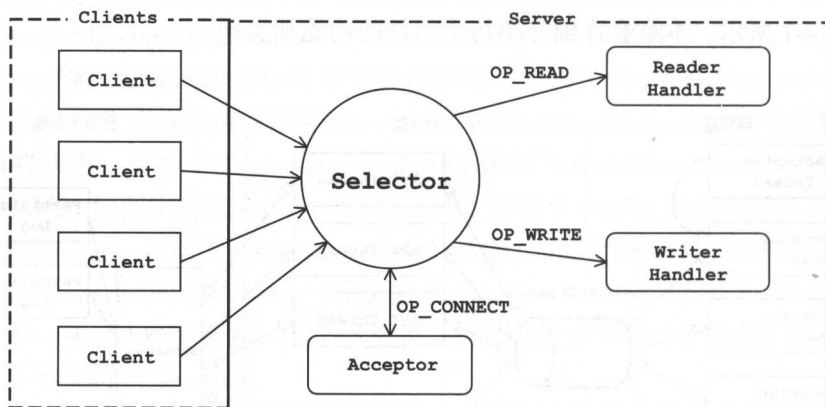


图 4-2

简单介绍其工作原理：

- (1) 首先创建 `ServerSocketChannel` 对象并在 `Selector` 上注册 `OP_ACCEPT` 事件，`ServerSocketChannel` 负责监听指定端口上的连接请求。
- (2) 当客户端发起到服务端的网络连接时，服务端的 `Selector` 监听到此 `OP_ACCEPT` 事件，会触发 `Acceptor` 来处理 `OP_ACCEPT`。
- (3) 当 `Acceptor` 接收到来自客户端的 `Socket` 连接请求时会为这个连接创建相应的 `SocketChannel`，将 `SocketChannel` 设置为非阻塞模式，并在 `Selector` 上注册其关注的 I/O 事件，

例如, OP_READ、OP_WRITE。此时, 客户端与服务端之间的 Socket 连接正式建立完成。

(4) 当客户端通过上面建立的 Socket 连接向服务端发送请求时, 服务端的 Selector 会监听到 OP_READ 事件, 并触发执行相应的处理逻辑(图 4-2 中的 Reader Handler)。当服务端可以向客户端写数据时, 服务端的 Selector 会监听到 OP_WRITE 事件, 并触发执行相应的处理逻辑(图 4-2 中的 Writer Handler)。

注意, 这里的所有事件处理逻辑都是在同一线程中完成的, 读者可以回顾 KafkaProducer 中的 Sender 线程以及 KafkaConsumer 的代码, 会发现它们都是这种设计。这种设计适合客户端这种并发连接数较小、数据量较小的场景, 对于服务端来说就有些缺点。例如, 某请求的处理过程比较复杂会造成线程阻塞, 那么所有后续请求都无法被处理, 这就会导致大量的请求超时。为了避免这种情况, 要求服务端在读取请求、处理请求以及发送响应等各个环节必须能迅速完成, 这就提高了编程难度, 在有的场景下是不可能完成的任务。而且这种模式不能利用服务器多核多处理器的并行处理能力, 造成了资源浪费。

为了满足高并发的需求, 也为了充分利用服务器的资源, 服务端需要使用多线程来执行业务逻辑。我们对上述架构稍作调整, 将网络读写的逻辑与业务处理的逻辑进行拆分, 让其由不同的线程池来处理, 从而实现多线程处理。设计架构如图 4-3 所示。

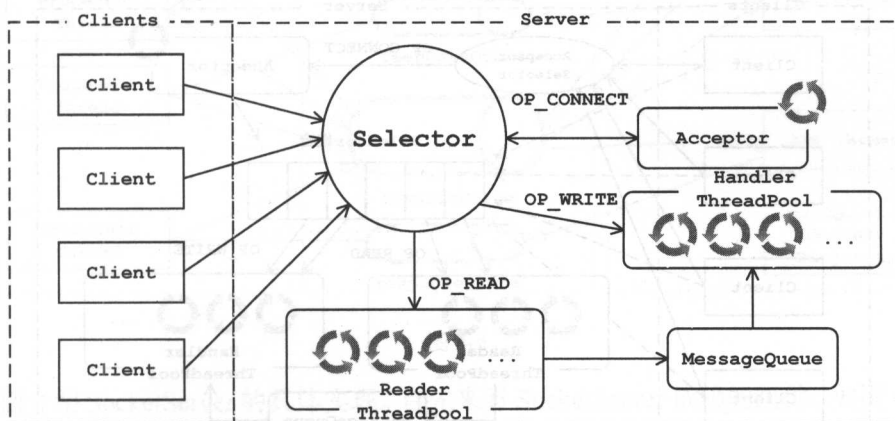


图 4-3

图 4-3 中的 Acceptor 单独运行在一个线程中, 也可以使用单线程的 ExecutorService 实现, 因为 ExecutorService 会在线程异常退出时, 创建新线程进行补偿, 所以可以防止出现线程异常退出后整个服务端不能接收请求的异常情况。Reader ThreadPool 线程池中的所有线程都会在 Selector 上注册 OP_READ 事件, 负责服务端读取请求的逻辑, 当然也是一个线程对应处理多个 Socket 连接。Reader ThreadPool 中的线程成功读取请求后, 将请求放入

MessageQueue 这个共享队列中。Handler ThreadPool 线程池中的线程会从 MessageQueue 中取出请求，然后执行业务逻辑对请求进行处理。这种模式下，即使处理某个请求的线程阻塞了，池中还有其他线程继续从 MessageQueue 中获取请求并进行处理，从而避免了整个服务端阻塞。当请求处理完成后，Handler 线程还负责产生响应并发送给客户端，这就要求 Handler ThreadPool 中的线程在 Selector 中注册 OP_WRITE 事件，实现发送响应的功能。

最后需要注意的是，当读取请求与业务处理之间的速度不匹配时，MessageQueue 队列长度的选择就显得尤为重要，尤其是 MessageQueue 队列是固定的大小的时候。如果队列长度太小，就会出现拒绝请求的情况；如果不限制 MessageQueue 队列的长度，则可能因为堆积过多未处理请求而导致内存溢出。这就需要设计人员根据实际的业务需求进行权衡和设计。

通过将网络处理与业务逻辑进行切分后实现了上述设计，此设计中读取、写入、业务处理都实现了多线程处理，不再存在性能瓶颈。但是，如果同一时间出现大量 I/O 事件，单个 Selector 就可能在分发事件时阻塞（或延时）而成为瓶颈。我们可以将上述设计中单独的 Selector 对象扩展成多个，让它们监听不同的 I/O 事件，这样就可以避免单个 Selector 带来的瓶颈问题。设计如图 4-4 所示。

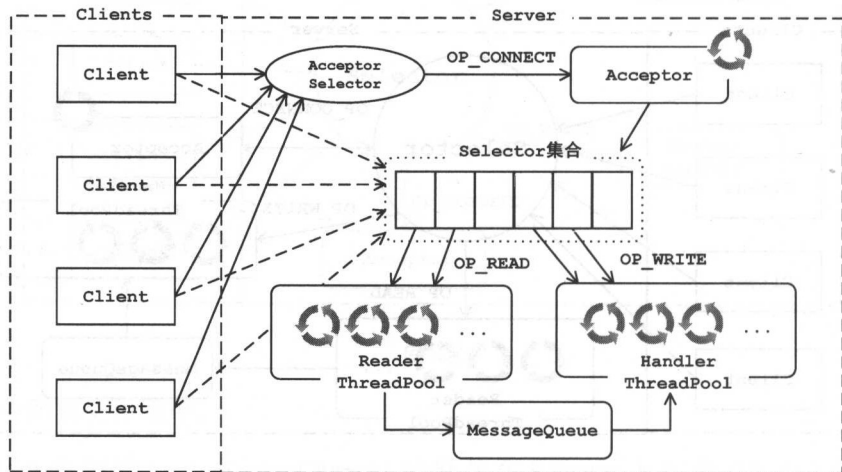


图 4-4

一般情况下，Acceptor 单独占用一个 Selector。当 Acceptor Selector 监听到 OP_ACCEPT 时，会创建相应的 SocketChannel，在图 4-4 的设计中，我们使用一定的策略，例如轮训 Selector 集合或选择注册连接数最少的 Selector，让不同的连接在不同的 Selector 上注册 I/O 事件。之后就由此 Selector 负责监听此 SocketChannel 上的事件。这样，就可以

缓解单个 Selector 带来的瓶颈问题。

4.1.2 SocketServer

Kafka 的网络层是采用多线程、多个 Selector 的设计实现的。核心类是 SocketServer，其中包含一个 Acceptor 用于接受并处理所有的新连接，每个 Acceptor 对应多个 Processor 线程，每个 Processor 线程拥有自己的 Selector，主要用于从连接中读取请求和写回响应。每个 Acceptor 对应多个 Handler 线程，主要用于处理请求并将产生响应返回给 Processor 线程。Processor 线程与 Handler 线程之间通过 RequestChannel 进行通信。整个网络层的结构如图 4-5 所示。

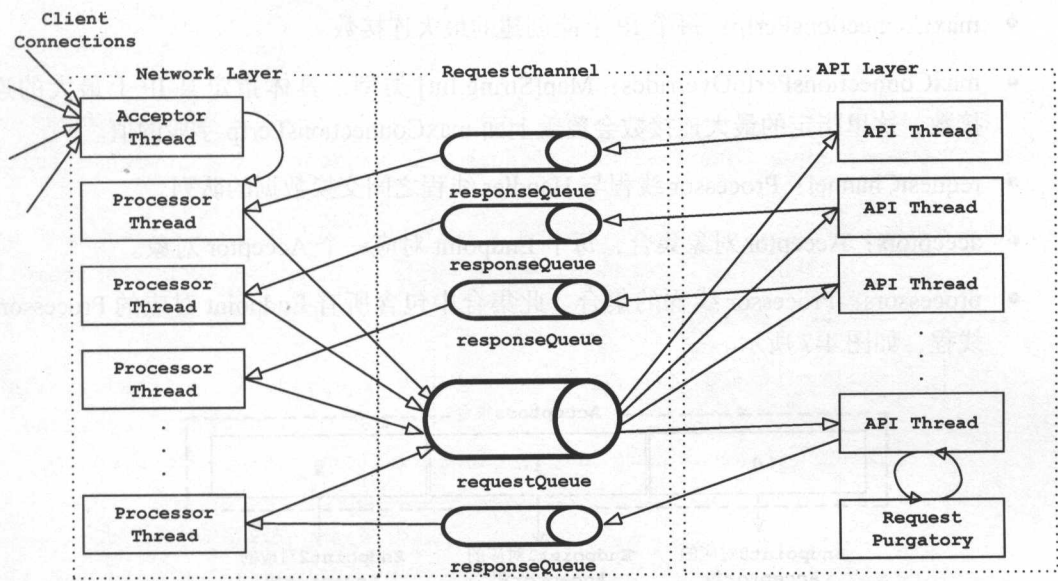


图 4-5

下面介绍 SocketServer 的具体实现。首先来看 SocketServer 依赖的组件，如图 4-6 所示。

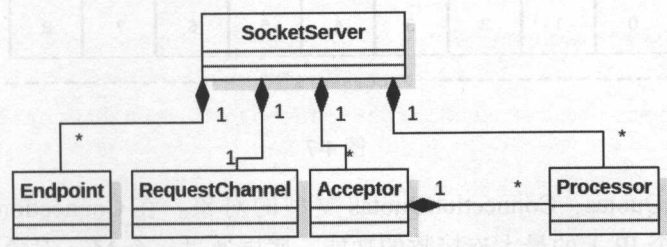


图 4-6

SocketServer 的核心字段如下所述。

- endpoints: Endpoint 集合。一般的服务器都有多块网卡，可以配置多个 IP，Kafka 可以同时监听多个端口。Endpoint 类中封装了需要监听的 host、port 及使用的网络协议。每个 Endpoint 都会创建一个对应的 Acceptor 对象。
- numProcessorThreads: Processor 线程的个数。
- totalProcessorThreads: Processor 线程的总个数，即 $\text{numProcessorThreads} * \text{endpoints.size}$ 。
- maxQueuedRequests: 在 RequestChannel 的 requestQueue 中缓存的最大请求个数。
- maxConnectionsPerIp: 每个 IP 上能创建的最大连接数。
- maxConnectionsPerIpOverrides: Map[String,Int] 类型，具体指定某 IP 上最大的连接数，这里指定的最大连接数会覆盖上面 maxConnectionsPerIp 字段的值。
- requestChannel: Processor 线程与 Handler 线程之间交换数据的队列。
- acceptors: Acceptor 对象集合，每个 Endpoint 对应一个 Acceptor 对象。
- processors: Processor 线程的集合。此集合中包含所有 Endpoint 对应的 Processors 线程，如图 4-7 所示。

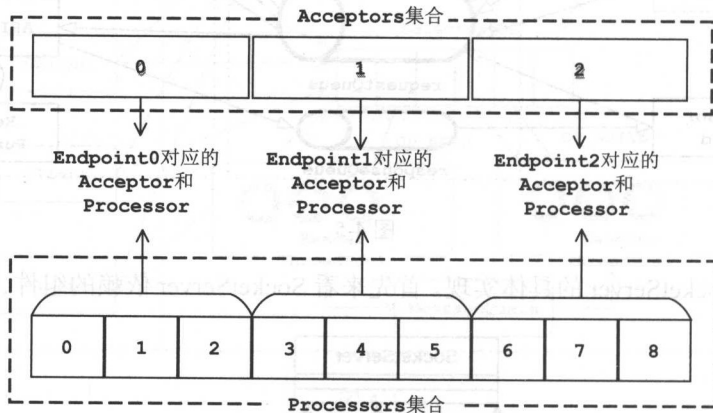


图 4-7

- connectionQuotas: ConnectionQuotas 类型的对象。在 ConnectionQuotas 中，提供了控制每个 IP 上的最大连接数的功能。底层通过一个 Map 对象，记录每个 IP 地址上建立的连接数，创建新 Connect 时与 maxConnectionsPerIpOverrides 指定的

最大值（或 `maxConnectionsPerIp`）进行比较，若超出限制，则报错。因为有多多个 `Acceptor` 线程并发访问底层的 `Map` 对象，则需要 `synchronized` 进行同步。

介绍完 `SocketServer` 中各个字段的功能，再来看一下 `SocketServer` 的初始化流程，读者可以先大体了解此过程，具体每个组件的实现在后面会详细介绍。`SocketServer` 在初始化时会创建遍历所有的 `Endpoint`，创建与其对应的 `Acceptor` 和 `Processor` 集合。

```
// 创建 RequestChannel，其中有 totalProcessorThreads 个 responseQueue 队列
val requestChannel = new RequestChannel(totalProcessorThreads,
maxQueuedRequests)

// 创建保存 Processor 的数组，长度为 totalProcessorThreads
private val processors = new Array[Processor](totalProcessorThreads)

// 创建保存 Acceptor 的集合
private[network] val acceptors = mutable.Map[EndPoint, Acceptor]()

// 向 RequestChannel 中添加一个监听器。此监听器实现的功能是：当 Handler 线程向某个
// responseQueue 中写入数据时，会唤醒对应的 Processor 线程进行处理
requestChannel.addResponseListener(id => processors(id).wakeup())

def startup() { // startup() 方法是 SocketServer 初始化的核心代码
  this.synchronized { // 同步
    connectionQuotas = new ConnectionQuotas(maxConnectionsPerIp,
      maxConnectionsPerIpOverrides) // 创建 ConnectionQuotas
    // Socket 的 sendBuffer 大小
    val sendBufferSize = config.socketSendBufferBytes
    // Socket 的 receiveBuffer 大小
    val recvBufferSize = config.socketReceiveBufferBytes
    var processorBeginIndex = 0

    endpoints.values.foreach { endpoint => // 遍历 Endpoints 集合
      val protocol = endpoint.protocolType
      val processorEndIndex = processorBeginIndex + numProcessorThreads

      // processors 数组从 processorBeginIndex~processorEndIndex，都是当前
      // Endpoint 对应的 Processor 对象的集合
      for (i <- processorBeginIndex until processorEndIndex) {
```

```

        // 创建 Processor 对象
        processors(i) = newProcessor(i, connectionQuotas, protocol)
    }

    // 创建 Acceptor, 同时为 processor 创建对应的线程。注意, 第五个参数指定了
    // processors 数组中与此 Acceptor 对象对应的 Processor 对象
    val acceptor = new Acceptor(endpoint, sendBufferSize, recvBufferSize,
brokerId, processors.slice(processorBeginIndex, processorEndIndex),
connectionQuotas)
    acceptors.put(endpoint, acceptor)
    // 创建 Acceptor 对应的线程, 并启动
    Utils.newThread("kafka-socket-acceptor-%s-%d".format(protocol.
toString, endpoint.port), acceptor, false).start()

    acceptor.awaitStartup() // 主线程阻塞等待 Acceptor 线程启动完成
    // 修改 processorBeginIndex, 为下一个 Endpoint 准备
    processorBeginIndex = processorEndIndex
}
}
}

```

SocketServer 的关闭操作比较简单, 它会关闭所有的 Acceptor 和 Processor, 代码如下:

```

def shutdown() = {
    this.synchronized { // 同步
        acceptors.values.foreach(_.shutdown) // 调用所有 Acceptor 的 shutdown
        processors.foreach(_.shutdown) // 调用所有 Processor 的 shutdown
    }
}

```

4.1.3 AbstractServerThread

Acceptor 和 Processor 都继承了 AbstractServerThread, 如图 4-8 所示, AbstractServerThread 是实现了 Runnable 接口的抽象类。在 AbstractServerThread 中为 Acceptor 和 Processor 提供了一些启动关闭相关的控制类方法。

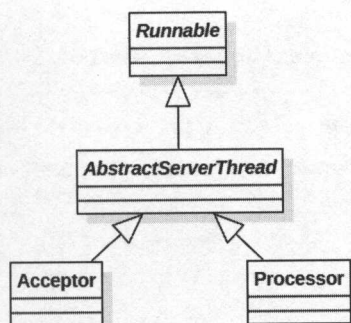


图 4-8

AbstractServerThread 中的关键字段有四个。

- **alive**: 标识当前线程是否存活，在初始化时设置为 true，在 shutdown() 方法中会将 alive 设置为 false。
- **shutdownLatch**: count 为 1 的 CountDownLatch 对象，标识了当前线程的 shutdown 操作是否完成。
- **startupLatch**: count 为 1 的 CountDownLatch 对象，标识了当前线程的 startup 操作是否完成。
- 在 awaitStartup() 和 shutdown() 方法中会调用 CountDownLatch.await() 方法，阻塞等待启动和关闭操作操作完成。在 startupComplete() 和 shutdownComplete() 方法中调用 CountDownLatch.countDown() 方法，唤醒阻塞的线程。
- **connectionQuotas**: 在 close() 方法中，根据传入的 ConnectionId，关闭 SocketChannel 并减少 ConnectionQuotas 中记录的连接数。

AbstractServerThread 中的方法主要是操作上面的四个字段，简单分析一下其中比较常用的几个方法：


```

// 阻塞等待启动操作完成
def awaitStartup(): Unit = startupLatch.await

// 标识启动操作完成, 唤醒阻塞线程
protected def startupComplete() = startupLatch.countDown()

// 抽象方法, 由子类实现
def wakeup()

def shutdown(): Unit = {
    alive.set(false) // 修改运行状态
    wakeup() // 唤醒当前 AbstractServerThread
    shutdownLatch.await() // 阻塞等待关闭操作完成
}

// 标识关闭操作完成, 唤醒阻塞线程
protected def shutdownComplete() = shutdownLatch.countDown()

// 关闭指定连接
def close(selector: KSelector, connectionId: String) {
    // 找到 connectionId 对应的连接
    val channel = selector.channel(connectionId)
    if (channel != null) {
        val address = channel.socketAddress
        if (address != null)
            connectionQuotas.dec(address) // 修改 connectionQuotas 记录的连接数
        selector.close(connectionId) // 关闭连接
    }
}

```

4.1.4 Acceptor

Acceptor 的主要功能是接收客户端建立连接的请求, 创建 Socket 连接并分配给 Processor 处理。Acceptor 中有两个比较重要的字段: 一个是 Java NIO Selector, 注意不要与第 2 章介绍的 KSelector 混淆; 二是用于接收客户端请求的 ServerSocketChannel 对象。在创建 Acceptor 时会初始化上面两个字段, 同时还会创建并启动其管理的 Processors 线程。

```
// 创建 nioSelector
private val nioSelector = NSelector.open()

// 创建 ServerSocketChannel
val serverChannel = openServerSocket(endPoint.host, endPoint.port)

this.synchronized { // 同步
    // 为其对应的每个 Processor 都创建对应的线程并启动
    processors.foreach { processor =>
        Utils.newThread("kafka-network-thread-%d-%s-%d".format(brokerId,
            endPoint.protocolType.toString, processor.id), processor, false).
start()
    }
}
```

Acceptor.run() 方法是 Acceptor 的核心逻辑，其中完成了对 OP_ACCEPT 事件的处理。具体实现如下：

```
def run() {
    // 注册 OP_ACCEPT 事件
    serverChannel.register(nioSelector, SelectionKey.OP_ACCEPT)
    startupComplete() // 标识当前线程启动操作已经完成
    try {
        var currentProcessor = 0
        while (isRunning) { // 检测线程运行状态
            try {
                val ready = nioSelector.select(500) // 等待关注的事件
                if (ready > 0) {
                    val keys = nioSelector.selectedKeys()
                    val iter = keys.iterator()
                    while (iter.hasNext && isRunning) {
                        try {
                            val key = iter.next
                            iter.remove()
                            if (key.isAcceptable) // 调用 accept() 方法处理 OP_ACCEPT 事件
                                accept(key, processors(currentProcessor))
                            else // 若不是 OP_ACCEPT 事件，则报错
                                throw new IllegalStateException("...")
                        }
                    }
                }
            }
        }
    }
```

```

        // 更新 currentProcessor, 从这里看出, 使用 Round-Robin 的方式选择 Processor
        currentProcessor = (currentProcessor + 1) % processors.length
    } catch {
        case e: Throwable => error("Error while accepting connection", e)
    }
}
}
} catch {
    // 异常处理 (略)
}
} finally {
    // 异常处理和日志输出 (略)
    shutdownComplete() // 标识此线程的关闭操作已完成
}
}

```

`Acceptor.accept()` 方法实现了对 `OP_ACCEPT` 事件的处理, 它会创建 `SocketChannel` 并将其交给 `Processor.accept()` 方法处理, 同时还会增加 `ConnectionQuotas` 中记录的连接数。`accept()` 方法的代码如下:

```

def accept(key: SelectionKey, processor: Processor) {
    val serverSocketChannel = key.channel().asInstanceOf[ServerSocketChannel]
    val socketChannel = serverSocketChannel.accept() // 创建 SocketChannel
    try {
        // 增加 ConnectionQuotas 中记录的连接数
        connectionQuotas.inc(socketChannel.socket().getInetAddress)

        // 配置 SocketChannel 的相关属性, 例如 sendBufferSize、keepalive 等
        socketChannel.configureBlocking(false)
        socketChannel.socket().setTcpNoDelay(true)
        socketChannel.socket().setKeepAlive(true)
        socketChannel.socket().setSendBufferSize(sendBufferSize)
        processor.accept(socketChannel) // 将 SocketChannel 交给 Processors 处理
    } catch {
        // 异常处理 (略)
        close(socketChannel) // 关闭 socketChannel
    }
}
}

```


4.1.5 Processor

Processor 主要用于完成读取请求和写回响应的操作，Processor 不参与具体业务逻辑的处理。Processor 的核心字段如下所述，在创建 Processor 对象时会初始化这些字段。

- `newConnections`: `ConcurrentLinkedQueue[SocketChannel]` 类型，其中保存了由此 Processor 处理的新建的 `SocketChannel`。
- `inflightResponses`: 保存未发送的响应。有读者可能会将 `inflightResponses` 与客户端的 `InFlightRequests` 进行类比，但也要注意其区别，客户端并不会对服务端发送的响应消息再次发送确认，所以 `inflightResponse` 中的响应会在发送成功后移除，而 `InFlightRequests` 中的请求是在收到响应后才移除。
- `selector`: `KSelector` 类型，负责管理网络连接。`KSelector` 的实现在第2章已经介绍过了，不再赘述。
- `requestChannel`: Processor 与 Handler 线程之间传递数据的队列。

在 `Acceptor.accept()` 方法中创建的 `SocketChannel` 会通过 `Processor.accept()` 方法交给 Processor 进行处理。`Processor.accept()` 方法接收到一个新的 `SocketChannel` 时会先将其放入 `newConnections` 队列中，然后会唤醒 Processor 线程来处理 `newConnections` 队列。注意，`newConnections` 队列由 Acceptor 线程和 Processor 线程并发操作，所以选择线程安全的 `ConcurrentLinkedQueue`。下面是 `accept()` 方法的代码：

```
def accept(socketChannel: SocketChannel) {  
    // 将 SocketChannel 添加到 newConnections 队列中  
    newConnections.add(socketChannel)  
  
    // Processor.wakeup() 方法通过调用 KSelector.wakeup() 方法实现，最终调用底层的  
    // Java NIO Selector 的 wakeup() 方法  
    wakeup()  
}
```

在 `Processor.run()` 方法中实现了从网络连接上读写数据的功能。`run()` 方法的流程如图 4-9 所示。

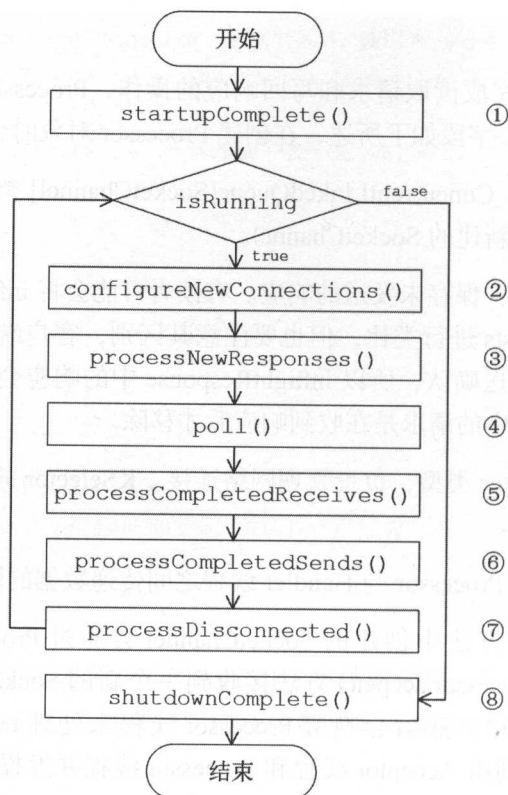


图 4-9

(1) 首先调用 `startupComplete()` 方法，标识 Processor 的初始化流程已经结束，唤醒阻塞等待此 Processor 初始化完成的线程。

(2) 处理 `newConnections` 队列中的新建 `SocketChannel`。队列中的每个 `SocketChannel` 都要在 `nioSelector` 上注册 `OP_READ` 事件。这里有个细节，`SocketChannel` 会被封装成 `KafkaChannel`，并附加 (`attach`) 到 `SelectionKey` 上，所以后面触发 `OP_READ` 事件时，从 `SelectionKey` 上获取的是 `KafkaChannel` 类型的对象。`KSelector` 以及 `KafkaChannel` 的相关介绍请读者参考第 2 章。下面是 `configureNewConnections()` 方法的代码：


```
private def configureNewConnections() {
  while (!newConnections.isEmpty) { // 遍历 newConnections 队列
    val channel = newConnections.poll()
    try {
      val localhost = channel.socket().getLocalAddress.getHostAddress
      val localPort = channel.socket().getLocalPort
      val remoteHost = channel.socket().getInetAddress.getHostAddress
      val remotePort = channel.socket().getPort
      // 根据 localhost、localPort、remoteHost、remotePort 的获取创建 ConnectionId
      val connectionId = ConnectionId(localhost, localPort,
        remoteHost, remotePort).toString

      selector.register(connectionId, channel) // 注册 OP_READ 事件
    } catch {
      ... ..// 异常处理 (略)
    }
  }
}
```

(3) 获取 RequestChannel 中对应的 responseQueue 队列，并处理其中缓存的 Response。

如果 Response 是 SendAction 类型，表示该 Response 需要发送给客户端，则查找对应的 KafkaChannel，为其注册 OP_WRITE 事件，并将 KafkaChannel.send 字段指向待发送的 Response 对象。同时还会将 Response 从 responseQueue 队列中移出，放入 inflightResponses 中。如果读者关心 OP_WRITE 事件的取消时机，可以回顾 KafkaChannel.send() 方法，即发送完一个完整的响应后，会取消此连接注册的 OP_WRITE 事件。

如果 Response 是 NoOpAction 类型，表示此连接暂无响应需要发送，则为 KafkaChannel 注册 OP_READ，允许其继续读取请求。

如果 Response 是 CloseConnectionAction 类型，则关闭对应的连接。

下面是 processNewResponses() 方法的代码：

```
private def processNewResponses() {
  // 在 RequestChannel 中使用 Processor 的 Id 绑定与 responseQueue 的对应关系
  // RequestChannel 的实现后面详述
  // 获取对应 responseQueue 中的响应
  var curr = requestChannel.receiveResponse(id)
  while (curr != null) {
    try {
      curr.responseAction match {
        case RequestChannel.NoOpAction => // 没有响应需要发送给客户端
          selector.unmute(curr.request.connectionId) // 注册 OP_READ 事件
        case RequestChannel.SendAction => // 该响应需要发送给客户端
          // 调用 KSelector.send() 方法，并将响应放入 inflightResponses 队列缓存
          sendResponse(curr)
        case RequestChannel.CloseConnectionAction =>
          close(selector, curr.request.connectionId)
      }
    } finally {
      curr = requestChannel.receiveResponse(id) // 继续处理 responseQueue
    }
  }
}
```

(4) 调用 `SocketServer.poll()` 方法读取请求，发送响应。`poll()` 方法底层调用的是 `KSelector.poll()` 方法。

```
private def poll() {
  selector.poll(300)
  // 异常处理代码(略)
}
```

`KSelector.poll()` 方法每次调用都会将读取的请求、发送成功的请求以及断开的连接放入其 `completedReceives`、`completedSends`、`disconnected` 队列中等待处理，下面就处理进行相应的队列。

(5) 调用 `processCompletedReceives()` 方法处理 `KSelector.completedReceives` 队列。首先，遍历 `completedReceives`，将 `NetworkReceive`、`ProcessorId`、身份认证信息一起封装成 `RequestChannel.Request` 对象并放入 `RequestChannel.requestQueue` 队列中，等待 `Handler` 线程的后续处理。之后，取消对应 `KafkaChannel` 注册的 `OP_READ` 事件，表示在发送响应之前，

此连接不能再读取任何请求了。

```
private def processCompletedReceives() {
  // 遍历 KSelector.completedReceives 队列
  selector.completedReceives.asScala.foreach { receive =>
    try {
      // 获取请求对应的 KafkaChannel
      val channel = selector.channel(receive.source)
      // 创建 KafkaChannel 对应的 Session 对象，与权限控制相关，在后面详细介绍
      val session = RequestChannel.Session(
        new KafkaPrincipal(KafkaPrincipal.USER_TYPE,
          channel.principal.getName),
        channel.socketAddress)

      // 将 NetworkReceive、ProcessorId、身份认证信息封装成 RequestChannel.Request 对象
      val req = RequestChannel.Request(processor = id, connectionId =
        receive.source, session = session, buffer = receive.payload,
        startTimeMs = time.milliseconds, securityProtocol = protocol)

      // 将 RequestChannel.Request 放入 RequestChannel.requestQueue 队列中等待处理
      requestChannel.sendRequest(req)
      selector.mute(receive.source) // 取消注册的 OP_READ 事件，连接不再读取数据
    } catch {
      ... ..// 异常处理代码（略）
    }
  }
}
```

(6) 调用 processCompletedSends() 方法处理 KSelector.completedSends 队列。首先，将 inflightResponses 中保存的对应 Response 删除。之后，为对应连接重新注册 OP_READ 事件，允许从该连接读取数据。

```
private def processCompletedSends() {
  // 遍历 completedSends 队列
  selector.completedSends.asScala.foreach { send =>
    // 此响应已经发送出去，从 inflightResponses 中删除
    val resp = inflightResponses.remove(send.destination).getOrElse {
```



```

        throw new IllegalStateException("...")
    }
    selector.unmute(send.destination) // 注册 OP_READ 事件, 允许此连接继续读取数据
}
}

```

(7) 调用 `processDisconnected()` 方法处理 `KSelector.disconnected` 队列。先从 `inflightResponses` 中删除该连接对应的所有 `Response`。然后, 减少 `ConnectionQuotas` 中记录的连接数, 为后续的新建连接做准备。

```

private def processDisconnected() {
    // 遍历了 disconnected 队列
    selector.disconnected.asScala.foreach {connectionId=>
        val remoteHost = ConnectionId.fromString(connectionId).getOrElse {
            throw new IllegalStateException("...")
        }.remoteHost

        // 从 InflightResponses 中删除该连接对应的所有 Response
        inflightResponses.remove(connectionId)
            .foreach(_.request.updateRequestMetrics())

        // 减少 ConnectionQuotas 中对应记录的连接数
        connectionQuotas.dec(InetAddress.getByName(remoteHost))
    }
}

```

(8) 当调用 `SocketServer.shutdown()` 关闭整个 `SocketServer` 时, 将 `alive` 字段设置为 `false`, 上述循环结束。然后调用 `shutdownComplete()` 方法执行一系列关闭操作: 关闭 `Processor` 管理的全部连接, 减少 `ConnectionQuotas` 中记录的连接数量, 标识自身的关闭流程已经结束, 唤醒等待该 `Processor` 结束的线程。

介绍完 `Processor.run()` 方法的执行流程和每一步的具体实现后, 来看一下 `run()` 方法的代码:

```

override def run() {
    startupComplete() // 步骤 1: 标记 Processor 启动流程已经完成, 唤醒等待的线程
    while (isRunning) { // 检测 alive 字段标识的运行状态
        try {

```

```

// 步骤 2: 处理 newConnections 队列中的新建 SocketChannel, 队列中的每个
// SocketChannel 都要在 nioSelector 上注册 OP_READ 事件
configureNewConnections()

processNewResponses() // 步骤 3: 处理 RequestChannel 中缓存的响应

poll() // 步骤 4: 通过 KSelector.poll() 方法, 进行网络 I/O

processCompletedReceives() // 步骤 5: 处理 KSelector.completedReceives 队列

processCompletedSends() // 步骤 6: 处理 KSelector.completedSends 队列

processDisconnected() // 步骤 7: 处理 KSelector.disconnected 队列
} catch {
    // 异常处理 (略)
}
}
swallowError(closeAll())
shutdownComplete() // 步骤 8: 执行一系列关闭操作
}

```

4.1.6 RequestChannel

Processor 线程与 Handler 线程之间传递数据是通过 RequestChannel 完成的。在 RequestChannel 中包含了一个 requestQueue 队列和多个 responseQueues 队列, 每个 Processor 线程对应一个 responseQueue。Processor 线程将读取到的请求存入 requestQueue 中, Handler 线程从 requestQueue 队列中取出请求进行处理; Handler 线程处理请求产生的响应会存放到 Processor 对应的 responseQueue 中, Processor 线程从其对应的 responseQueue 中取出响应并发送给客户端。RequestChannel 的结构如图 4-10 所示。

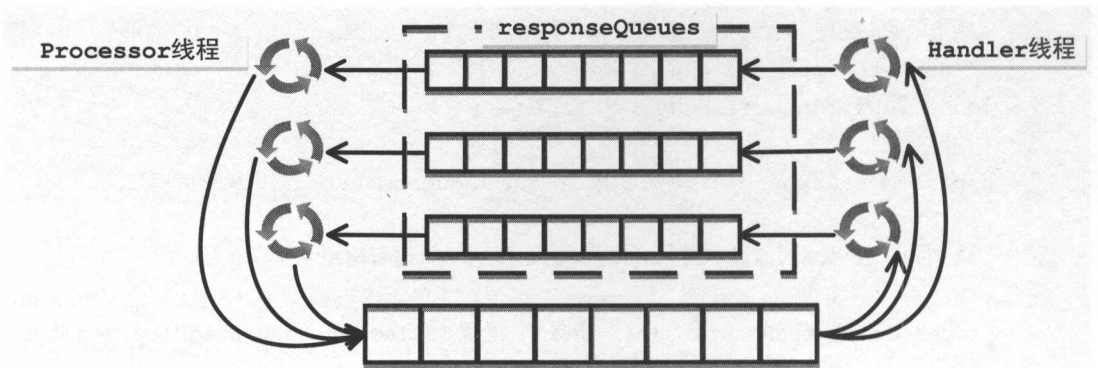


图 4-10

下面来看一下 RequestChannel 的核心字段。

- requestQueue: ArrayBlockingQueue[RequestChannel.Request] 类型。Processor 线程向 Handler 线程传递请求的队列。因为多个 Processor 线程和多个 Handler 线程并发操作，所以选择线程安全的队列。
- responseQueues: LinkedBlockingQueue [RequestChannel.Response] 队列的数组，Handler 线程向 Processor 线程传递响应的队列，每个 Processor 对应该数组中的一个队列。
- numProcessors: Processor 线程个数，也是 responseQueues 这个数组的长度。
- queueSize: 缓存请求的最大个数，即 requestQueue 的长度。
- responseListeners: List[(Int) => Unit] 类型，该字段是监听器列表，其中的监听器的主要作用是 Handler 线程向 responseQueue 存放响应时唤醒对应的 Processor 线程。

RequestChannel 提供了增删 requestQueue 队列、responseQueues 集合以及 responseListeners 列表中元素的方法。在 SocketServer 的初始化过程中，有向 RequestChannel.responseListeners 集合中添加一个唤醒对应 Processor 线程的监听，代码如下所示。

```
requestChannel.addResponseListener(id => processors(id).wakeup())
```

需要注意的是，每次向 responseQueues 添加请求时都要触发 responseListeners 列表中的监听器。具体实现如下：

```
// 向对应 responseQueue 队列中添加 SendAction 类型的 Response
def sendResponse(response: RequestChannel.Response) {
    responseQueues(response.processor).put(response)
    for (onResponse <- responseListeners) // 调用 responseListeners 集合中的监听器
        onResponse(response.processor)
}

// 向对应 responseQueue 队列中添加 NoOpAction 类型的 Response
def noOperation(processor: Int, request: RequestChannel.Request) {
    responseQueues(processor).put(new RequestChannel.Response(processor,
        request,
        null, RequestChannel.NoOpAction))
    for (onResponse <- responseListeners) // 调用 responseListeners 集合中的监听器
        onResponse(processor)
}

// 向对应 responseQueue 队列中添加 CloseConnectionAction 类型的 Response
def closeConnection(processor: Int, request: RequestChannel.Request) {
    responseQueues(processor).put(new RequestChannel.Response(processor,
        request, null, RequestChannel.CloseConnectionAction))
    for (onResponse <- responseListeners) // 调用 responseListeners 集合中的监听器
        onResponse(processor)
}
```

在 RequestChannel 中保存的是 RequestChannel.Request 和 RequestChannel.Response 两个类的对象。RequestChannel.Request 会对请求进行解析，形成 requestId（请求类型 ID）、header（请求头）、body（请求体）等字段，供 Handler 线程使用，并提供了一些记录操作时间的字段供监控程序使用。简单看一下 RequestChannel.Request 的解析过程：

```
case class Request(processor: Int, connectionId: String, session: Session,
    private var buffer: ByteBuffer, startTimeMs: Long,
    securityProtocol: SecurityProtocol) {
    // 下面是一些记录操作时间的字段，注意，由于这些字段存在跨线程的比较和修改，使用
    // @volatile 修饰，保证可见性
    @volatile var requestDequeueTimeMs = -1L
    @volatile var apiLocalCompleteTimeMs = -1L
    // 其他记录字段（略）
```

```

val requestId = buffer.getShort() // 请求类型 ID
val header: RequestHeader = // 请求头
{
    buffer.rewind
    try RequestHeader.parse(buffer) // 解析请求头
    catch {
        // 异常处理 (略)
    }
}

val body: AbstractRequest = // 请求体
{
    try {
        if (header.apiKey == ApiKeys.API_VERSIONS.id
            && !Protocol.apiVersionSupported(header.apiKey, header.
apiVersion))
            new ApiVersionsRequest
        else
            // 解析请求体
            AbstractRequest.getRequest(header.apiKey, header.apiVersion,
buffer)
    } catch {
        // 异常处理 (略)
    }
}

buffer = null // 已经解析完成, 将 buffer 置空, help GC

```

`RequestChannel.Response` 需要注意其 `responseAction` 字段, 有 `SendAction`、`NoOpAction`、`CloseConnectionAction` 三种类型, 在前面介绍 `Processor` 的 `processNewResponses()` 方法时也介绍过这三种类型的含义, 此处不再赘述。

介绍到这里, 有读者可能会问: 当请求放入 `RequestChannel.requestQueue` 之后, 会有多个 `Handler` 线程并发处理从其中取出请求处理, 那如何保证客户端请求的顺序性呢? 请读者返回前面查看 `Processor.run()` 方法的介绍, 其中有多处注册 / 取消 `OP_READ` 事件以及注册 / 取消 `OP_WRITE` 事件的操作, 通过这些操作的组合可以保证每个连接上只有一个请求和一个对应的响应, 从而实现请求的顺序性。

现在回头来总结一个请求数据从生产者发送到服务端的流转过程, 如图 4-11 所示。

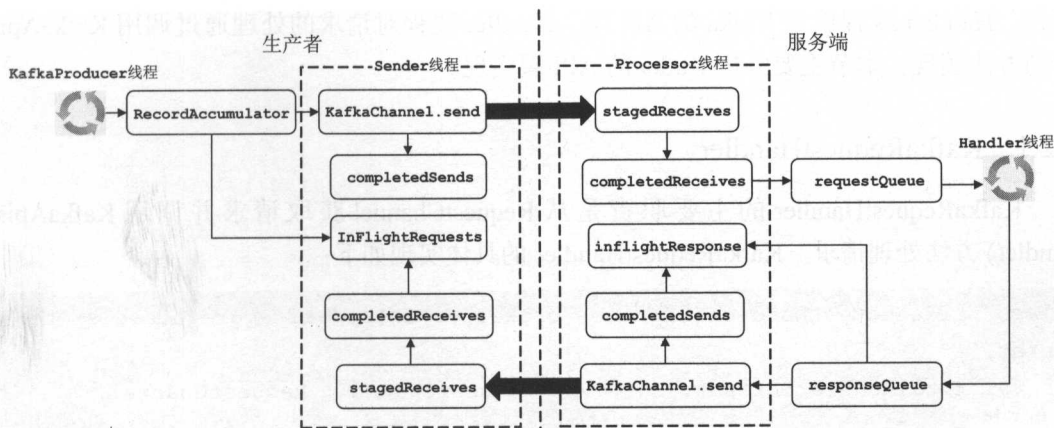


图 4-11

KafkaProducer 线程创建 ProducerRecord 后，会将其缓存进 RecordAccumulator。Sender 线程从 RecordAccumulator 中获取缓存的消息，放入 KafkaChannel.send 字段中等待发送，同时放入 InFlightRequests 队列中等待响应。之后，客户端会通过 KSelector 将请求发送出去。在服务端，Processor 线程使用 KSelector 读取请求并暂存到 stageReceives 队列中，KSelector.poll() 方法结束后，请求被移转移到 completeReceives 队列中。之后，Processor 将请求进行一些解析操作后，放入 RequestChannel.requestQueue 队列。Handler 线程会从 RequestChannel.requestQueue 队列中取出请求进行处理，将处理之后生成的响应放入 RequestChannel.responseQueue 队列。Processor 线程从其对应的 RequestChannel.responseQueue 队列中取出响应并放入 inflightResponses 队列中缓存，当响应发送出去之后会将其从 inflightResponse 中删除。生产者读取响应的过程与服务端读取请求的过程类似，主要的区别是生产者需要对 InFlightRequest 中的请求进行确认。消费者与服务端之间的请求和响应的流转过程与上述过程类似，不再赘述了。

Kafka 网络层的设计原理和实现就介绍到这里了。在高性能的分布式框架中经常采用这种 Reactor 模式的设计，例如，HDFS RPC 框架的服务端、ZooKeeper 等。也有实现了 Reactor 模式的框架，例如，Netty 和 Mina。希望读者能够通过本章的描述理解 Reactor 模式及其在 Kafka 服务端的实现。

4.2 API 层

通过上一节对 Kafka 网络层的介绍我们知道，Handler 线程会取出 Processor 线程，放入 RequestChannel 的请求进行处理，并将产生的响应通过 RequestChannel 传递给 Processor

线程。Handler 线程属于 Kafka 的 API 层，Handler 线程对请求的处理通过调用 KafkaApis 中的方法实现。本节主要分析 Kafka 的 API 层实现。

4.2.1 KafkaRequestHandler

KafkaRequestHandler 的主要职责是从 RequestChannel 获取请求并调用 KafkaApis.handle() 方法处理请求。KafkaRequestHandler 的具体实现如下：

```
class KafkaRequestHandler(id: Int, brokerId: Int, val aggregateIdleMeter:
Meter,
    val totalHandlerThreads: Int, val requestChannel: RequestChannel,
    apis: KafkaApis) extends Runnable with Logging {

    def run() {
        while (true) {
            try {
                var req: RequestChannel.Request = null
                while (req == null) {
                    val startSelectTime = SystemTime.nanoseconds

                    // 从 RequestChannel.requestQueue 获取请求，通过 requestQueue.poll() 方法实现
                    req = requestChannel.receiveRequest(300)
                    val idleTime = SystemTime.nanoseconds - startSelectTime
                    // 统计监控指标，后面详述
                    aggregateIdleMeter.mark(idleTime / totalHandlerThreads)
                }

                // 读取到 RequestChannel.AllDone 请求，KafkaRequestHandler 线程结束
                if (req eq RequestChannel.AllDone) {
                    return
                }

                // KafkaApis 类中实现了处理请求的逻辑，KafkaApis 还负责将响应写回对应的
                // RequestChannel.responseQueue 中，唤醒 Processor 处理
                apis.handle(req)
            } catch {
                case e: Throwable => error("Exception when handling request", e)
            }
        }
    }
}
```

```

    }
  }
}

def shutdown(): Unit =
// 发送 RequestChannel.AllDone
requestChannel.sendRequest(RequestChannel.AllDone)
}

```

API 层使用 `KafkaRequestHandlerPool` 来管理所有的 `KafkaRequestHandler` 线程, `KafkaRequestHandlerPool` 是一个简易版的线程池, 其中创建了多个 `KafkaRequestHandler` 线程。 `KafkaRequestHandlerPool` 的代码如下:

```

class KafkaRequestHandlerPool(val brokerId: Int, val requestChannel:
RequestChannel,
    val apis: KafkaApis, numThreads: Int) extends Logging with
KafkaMetricsGroup {

  // 用于保存执行 KafkaRequestHandler 的线程
  val threads = new Array[Thread](numThreads)

  // KafkaRequestHandler 集合
  val runnables = new Array[KafkaRequestHandler](numThreads)
  for (i <- 0 until numThreads) {
    // 创建 KafkaRequestHandler 对象及其对应线程
    runnables(i) = new KafkaRequestHandler(i, brokerId, aggregateIdleMeter,
numThreads, requestChannel, apis)
    threads(i) = Utils.daemonThread("kafka-request-handler-" + i, runnables(i))
    threads(i).start() // 启动线程
  }

  def shutdown() {
    for (handler <- runnables)
handler.shutdown // 停止所有 KafkaRequestHandler 线程

    for (thread <- threads)
thread.join // 阻塞等待所有 KafkaRequestHandler 线程结束
  }
}

```

4.2.2 KafkaApis

KafkaApis 是 Kafka 服务器处理请求的入口类。它负责将 KafkaRequestHandler 传递过来的请求分发到不同的 `handle*()` 处理方法中，分发的依据是 `RequestChannel.Request` 中的 `requestId`，此字段保存了请求的 `ApiKeys` 的值，不同的 `ApiKeys` 值表示不同请求的类型。下面来看一下实现了此分发功能 `handle()` 方法的具体实现：

```
def handle(request: RequestChannel.Request) {
  try {
    ApiKeys.forId(request.requestId) match { // 根据 requestId 来分发请求

      // ApiKeys.PRODUCE 表示 ProducerRequest，具体格式请读者参考第 2 章的介绍
      // ProducerRequest 会交由 handleProducerRequest() 方法进行处理
      case ApiKeys.PRODUCE => handleProducerRequest(request)

      // ApiKeys.FETCH 表示 ProducerRequest，具体格式请读者参考第 2 章的介绍
      // 这里将 FetchRequest 交由 handleFetchRequest() 方法进行处理
      case ApiKeys.FETCH => handleFetchRequest(request)

      // ApiKeys.METADATA 表示 MetadataRequest，请读者参考第 2 章的相关内容
      // 这里将 MetadataRequest 交由 handleTopicMetadataRequest() 方法进行处理
      case ApiKeys.METADATA => handleTopicMetadataRequest(request)

      ..... // 后面还有很多 ApiKeys 类型以及对应的 handle*() 方法，篇幅限制，不再一一列举
    }
  } catch {
    // 异常处理代码（略）
  }
}
```

本章不再一一列举每个 `handle*()` 方法，我们将在后面介绍 Kafka 各个子系统时穿插相关请求的 `handle*()` 处理方法。

4.3 日志存储

4.3.1 基本概念

首先需要了解的是, Kafka 使用日志文件的方式保存生产者发送的消息。每条消息都有一个 offset 值来表示它在分区中的偏移量, 这个 offset 值是逻辑值, 并不是消息实际存放的物理地址。offset 值类似于数据库表中的主键, 主键唯一确定了数据库表中的一条记录, offset 唯一确定了分区中的一条消息。Kafka 存储机制在逻辑上如图 4-12 所示。



图 4-12

为了提高写入的性能, 同一个分区中的消息是顺序写入的, 这就避免了随机写入带来的性能问题。在第 1 章介绍 Kafka 相关概念时提到过, 一个 Topic 可以划分成多个分区, 而每个分区又有多个副本。当一个分区的副本（无论是 Leader 副本还是 Follower 副本）被划分到某个 Broker 上时, Kafka 就要在此 Broker 上为此分区建立相应的 Log, 而生产者发送的消息会存储在 Log 中, 供消费者拉取后消费。

Kafka 中存储的一般都是海量消息数据, 为了避免日志文件太大, Log 并不是直接对应于磁盘上的一个日志文件, 而是对应磁盘上的一个目录, 这个目录的命名规则是 `<topic_name>_<partition_id>`, Log 与分区之间的关系是一一对应的, 对应分区中的全部消息都存储在此目录下的日志文件中。

Kafka 通过分段的方式将 Log 分为多个 LogSegment, LogSegment 是一个逻辑上的概念, 一个 LogSegment 对应磁盘上的一个日志文件和一个索引文件, 其中日志文件用于记录消息, 索引文件中保存了消息的索引。随着消息的不断写入, 日志文件的大小到达一个阈值时, 就创建新的日志文件和索引文件继续写入后续的消息和索引信息。日志文件的文件名的命名规则是 `[baseOffset].log`, baseOffset 是日志文件中第一条消息的 offset。图 4-13 展示了一个 Log 的结构。

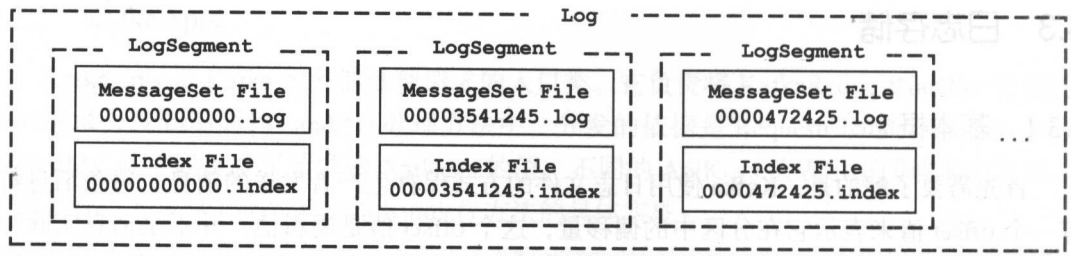


图 4-13

为了提高查询消息的效率，每个日志文件都对应一个索引文件，这个索引文件并没有为每条消息都建立索引项，而是使用稀疏索引方式为日志文件中的部分消息建立了索引。图 4-14 展示了索引文件与日志文件之间的对应关系。

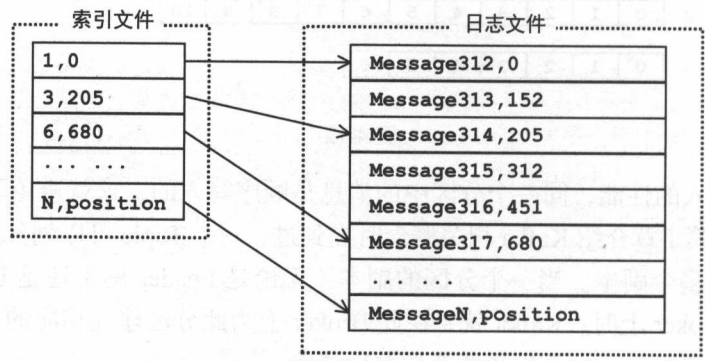


图 4-14

介绍完 Kafka 日志存储的基本概念之后，下面来分析实现日志存储功能的相关代码。

4.3.2 FileMessageSet

在 Kafka 中使用 FileMessageSet 管理上文介绍的日志文件，它对应磁盘上的一个真正的日志文件。FileMessageSet 继承了 MessageSet 抽象类，如图 4-15 (a) 所示。MessageSet 中保存的数据格式分为三部分，如图 4-15 (b) 所示：8 字节的 offset 值，4 字节的 size 表示 message data 大小，这两部分组合成为 LogOverhead，message data 部分保存了消息的数据，逻辑上对应一个 Message 对象。

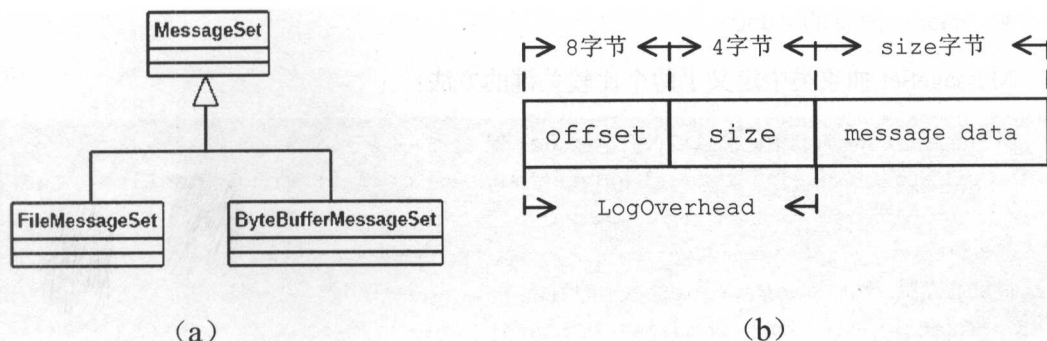


图 4-15

Kafka 使用 `Message` 类表示消息，`Message` 使用 `ByteBuffer` 保存数据，其格式及各个部分的含义如图 4-16 所示。

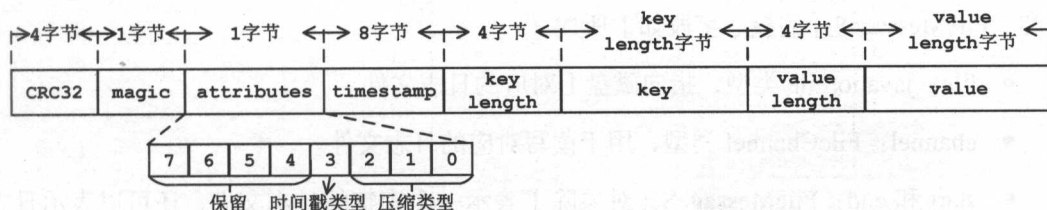


图 4-16

- `CRC32`: 4 个字节，消息的校验码。
- `magic`: 1 字节，魔数标识，与消息格式有关，取值为 0 或 1。当 `magic` 为 0 时，消息的 `offset` 使用绝对 `offset` 且消息格式中没有 `timestamp` 部分；当 `magic` 为 1 时，消息的 `offset` 使用相对 `offset` 且消息格式中存在 `timestamp` 部分。所以，`magic` 值不同，消息的长度是不同的。
- `attributes`: 1 字节，消息的属性。其中第 0 ~ 2 位的组合表示消息使用的压缩类型，0 表示无压缩，1 表示 `gzip` 压缩，2 表示 `snappy` 压缩，3 表示 `lz4` 压缩。第 3 位表示时间戳类型，0 表示创建时间，1 表示追加时间。
- `timestamp`: 时间戳，其含义由 `attribute` 的第 3 位确定。
- `key length`: 消息 `key` 的长度。
- `key`: 消息的 `key`。
- `value length`: 消息的 `value` 长度。

- value: 消息的 value。

MessageSet 抽象类中定义了两个比较关键的方法:

```
// 将当前 MessageSet 中的消息写入到 Channel 中
def writeTo(channel: GatheringByteChannel, offset: Long, maxSize: Int):
Int

// 提供迭代器, 顺序读取 MessageSet 中的消息
def iterator: Iterator[MessageAndOffset]
```

这两个方法说明 MessageSet 具有顺序写入消息和顺序读取的特性。在后面对 FileMessageSet 和 ByteBufferMessageSet 的介绍过程中会介绍这两个方法的实现。

了解了 MessageSet 抽象类以及其中保存消息的格式, 我们开始分析 FileMessageSet 实现类。FileMessageSet 的核心字段如下所述。

- file: java.io.File 类型, 指向磁盘上对应的日志文件。
- channel: FileChannel 类型, 用于读写对应的日志文件。
- start 和 end: FileMessageSet 对象除了表示一个完整的日志文件, 还可以表示日志文件分片 (Slice), start 和 end 表示分片的起始位置和结束位置。文件分配的相关概念不做详细介绍, 请读者参考相关资料。
- isSlice: Boolean 类型, 表示当前 FileMessageSet 是否为日志文件的分片。
- _size: FileMessageSet 大小, 单位是字节。如果 FileMessageSet 是日志文件的分片, 则表示分片大小 (即 end-start 的值); 如果不是分片, 则表示整个日志文件的大小。注意, 因为可能有多个 Handler 线程并发向同一个分区写入消息, 所有 _size 是 AtomicInteger 类型。

在 FileMessageSet 中有多个重载的构造方法, 这里选择一个比较重要的方法进行介绍。此构造方法会创建一个非分片的 FileMessageSet 对象。在 Window NTFS 文件系统以及老版本的 Linux 文件系统中, 进行文件的预分配会提高后续写操作的性能, 为此 FileMessageSet 提供了 preallocate 的选项, 决定是否开启预分配的功能。我们也可以通过 FileMessageSet 构造函数的 mutable 参数决定是否创建只读的 FileMessageSet。

```

def this(file: File, fileAlreadyExists: Boolean, initFileSize: Int,
preallocate: Boolean) =
    this(file,
// 需要注意的是, 如果使用 preallocate 进行预分配空间, end 会被初始化为 0
        channel = FileMessageSet.openChannel(file, mutable = true,
fileAlreadyExists, initFileSize, preallocate),
        start = 0,
        end = (if (!fileAlreadyExists && preallocate) 0 else Int.MaxValue),
        isSlice = false)
// 下面是 FileMessageSet.openChannel() 方法的具体实现
def openChannel(file: File, mutable: Boolean, fileAlreadyExists: Boolean =
false,
initFileSize: Int = 0, preallocate: Boolean = false): FileChannel = {

    if (mutable) { // 根据 mutable 参数创建的 FileChannel 是否可写
        if (fileAlreadyExists)
            new RandomAccessFile(file, "rw").getChannel()
        else {
            if (preallocate) { // 进行文件预分配
                val randomAccessFile = new RandomAccessFile(file, "rw")
                randomAccessFile.setLength(initFileSize)
                randomAccessFile.getChannel()
            } else
                new RandomAccessFile(file, "rw").getChannel() // 创建可读可写的
FileChannel
        }
    } else
        new FileInputStream(file).getChannel() // 创建只读的 FileChannel
    }
}

```

在 FileMessageSet 对象初始化的过程中, 会移动 FileChannel 的 position 指针, 这是为了实现每次写入的消息都在日志文件的尾部, 从而避免重启服务后的写入操作覆盖之前的操作。对于新创建的且进行了预分配空间的日志文件, 其 end 会初始化为 0, 所以也是从文件起始写入数据的。


```
// 将 position 移动到最后一个字节，之后就可以从此 position 开始写消息。注意，对于分片
// FileMessageSet 读者可以尝试跟踪一下代码，会发现执行写入操作
// (FileMessageSet.append() 方法) 的都是非分片的 FileMessageSet 对象
if (!isSlice)
    channel.position(math.min(channel.size.toInt, end))
```

介绍完 `FileMessageSet` 的构造过程，下面来分析其读写过程。`FileMessageSet.append()` 方法实现了写日志文件的功能，需要注意的是其参数必须是 `ByteBufferMessageSet` 对象，`ByteBufferMessageSet` 的内容后面介绍。下面是 `FileMessageSet.append()` 方法的代码：

```
def append(messages: ByteBufferMessageSet) {
    val written = messages.writeFullyTo(channel) // 写文件
    _size.getAndAdd(written) // 修改 FileMessageSet 的大小
}

// 下面是 ByteBufferMessageSet.writeFullyTo() 方法
def writeFullyTo(channel: GatheringByteChannel): Int = {
    buffer.mark()
    var written = 0
    while (written < sizeInBytes) // 将 ByteBufferMessageSet 中的数据全部写入文件
        written += channel.write(buffer)
    buffer.reset()
    written
}
```

查找指定消息的功能在 `FileMessageSet.searchFor()` 方法中实现。`searchFor()` 的逻辑是：从指定的 `startingPosition` 开始逐条遍历 `FileMessageSet` 中的消息，并将每个消息的 `offset` 与 `targetOffset` 进行比较，直到 `offset` 大于等于 `targetOffset`，最后返回查找到的 `offset`。在整个遍历过程中不会将消息的 `key` 和 `value` 读取到内存，而是只读取 `LogOverhead`（即 `offset` 和 `size`），并通过 `size` 定位到下一条消息的开始位置。`FileMessageSet.searchFor()` 方法的代码如下：

```

def searchFor(targetOffset: Long, startPosition: Int): OffsetPosition =
{
    var position = startPosition // 起始位置
    // 创建用于读取 LogOverhead (即 offset+size) 的 ByteBuffer (长度为 12)
    val buffer = ByteBuffer.allocate(MessageSet.LogOverhead)
    val size = sizeInBytes() // 当前 FileMessageSet 的大小, 单位是字节

    // 从 position 开始逐条消息遍历
    while (position + MessageSet.LogOverhead < size) {
        buffer.rewind() // 重置 ByteBuffer 的 position 指针, 准备读入数据

        // 读取 LogOverhead。这里会确保 startPosition 位于一个消息的开头, 否则读取到
        // 的并不是 LogOverhead, 这个条件的保证会在后面提到
        channel.read(buffer, position)
        if (buffer.hasRemaining()) // 未读取到 12 个字节的 LogOverhead, 抛出异常
            throw new IllegalStateException("...")

        // 重置 ByteBuffer 的 position 指针, 准备从 ByteBuffer 中读取数据
        buffer.rewind()
        val offset = buffer.getLong() // 获取消息的 offset, 8 个字节
        if (offset >= targetOffset) { // 判断是否符合退出条件
            // 这里将 offset 和对应的 position (物理地址) 封装成 OffsetPosition 这种对象后返回
            return OffsetPosition(offset, position) // Got You!
        }
        val messageSize = buffer.getInt() // 获取消息的 size, 4 个字节
        // 移动 position, 准备读取下个消息
        position += MessageSet.LogOverhead + messageSize
    }
    null // 找不到 offset 大于等于 targetOffset 的消息, 则返回 null
}

```

`FileMessageSet.writeTo()` 方法是将在 `FileMessageSet` 中的数据写入指定的其他 Channel 中, 这里先了解此方法的功能, 具体实现会在后面介绍“零复制”的时候一起介绍。`FileMessageSet.read*()` 方法是从 `FileMessageSet` 中读取数据, 可以将 `FileMessageSet` 中的数据读入到别的 `ByteBuffer` 中返回, 也可以按照指定位置和长度形成分片的 `FileMessageSet` 对象返回。`FileMessageSet.delete()` 方法是整个日志文件删除。

`FileMessageSet` 还有一个 `truncateTo()` 方法, 主要负责将日志文件截断到 `targetSize`

大小。此方法在后面介绍分区中 Leader 副本切换时还会提到。下面是 `truncateTo()` 方法的具体实现：

```
def truncateTo(targetSize: Int): Int = {
    val originalSize = sizeInBytes
    if (targetSize > originalSize || targetSize < 0) // 检测 targetSize 的有效性
        throw new KafkaException("...")
    if (targetSize < channel.size.toInt) {
        channel.truncate(targetSize) // 裁剪文件
        channel.position(targetSize) // 移动 position
        _size.set(targetSize) // 修改 _size
    }
    originalSize - targetSize // 返回裁剪掉的字节数
}
```

`FileMessageSet` 还实现了 `iterator()` 方法，返回一个迭代器。`FileMessageSet` 迭代器读取消息的逻辑是：先读取消息的 `LogOverhead` 部分，然后按照 `size` 分配合适的 `ByteBuffer`，再读取 `message data` 部分，最后将 `message data` 和 `offset` 封装成 `MessageOffset` 对象返回。此迭代器的实现与 `searchFor()` 方法类似，感兴趣的读者可以参考源码学习。

4.3.3 ByteBufferMessageSet

`MessageSet` 的另一个子类是 `ByteBufferMessageSet`，`FileMessageSet.append()` 方法的参数就是此类的对象。为什么必须 `append()` 方法的参数是 `ByteBufferMessageSet`，而不是直接使用 `ByteBuffer` 呢？

回顾一下，在第 2 章介绍 `MemoryRecords` 时提到，向 `MemoryRecords` 写入消息时，可以使用 `Compressor` 对消息批量进行压缩，然后将压缩后的消息发送给服务端。

在有些设计中，将每个请求的负载单独压缩后再进行传输，这种设计虽然可以减小传输的数据量，但是存在一个小问题，我们常见压缩算法是数据量越大压缩率越高，一般情况下每个请求的负载不会特别大，这就导致压缩率比较低。`Kafka` 实现的压缩方式是将多个消息一起进行压缩，这样就可以保证较高的压缩率。而且在一般情况下，生产者发送的压缩数据在服务端也是以保持压缩状态进行存储的，消费者从服务端获取的也是压缩消息，消费者在处理消息之前才会解压消息，这也就实现了“端到端的压缩”。

压缩后的消息格式与非压缩的消息格式类似，但是分为两层，如图 4-17 所示。

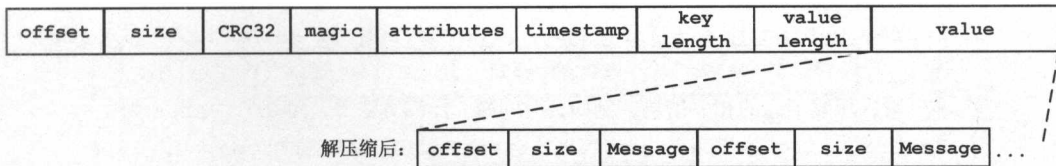


图 4-17

压缩消息的 key 为 null，所以图 4-17 没有画出 key 这部分；value 中保存的是多条消息压缩数据。

创建压缩消息

在开始先回头看第 2 章介绍的 Compressor 的构造函数的代码：

```
public Compressor(ByteBuffer buffer, CompressionType type) {
    // 初始化字段（略）
    if (type != CompressionType.NONE) {
        // 如果进行消息压缩，则在 buffer 的首部提前空出 offset 和 size 的空间 (Records.
        // LOG_OVERHEAD) 以及 CRC32 等字段的的空间 (Record.RECORD_OVERHEAD)
        buffer.position(initPos + Records.LOG_OVERHEAD + Record.RECORD_
OVERHEAD);
    }
    // 根据压缩算法类型，初始化 bufferStream 和 appendStream，第 2 章介绍过（略）
}
```

通过 MemoryRecords.append() 方法不断写入消息并压缩的过程在第 2 章已经分析过了，这里不再赘述。当 MemoryRecords 写满，则会调用 Compressor.close() 方法，完成 offset、size、CRC32 等字段的写入，之后就可以发送到服务端了。Compressor.close() 的实现如下：

```
public void close() {
    appendStream.close(); // 要压缩的消息已经写完了，先关闭压缩输出流
    if (type != CompressionType.NONE) { // 使用了压缩算法
        ByteBuffer buffer = bufferStream.buffer();
        int pos = buffer.position(); // 保存 buffer 尾部位置
        buffer.position(initPos); // 移动 position 指针到 buffer 头部

        buffer.putLong(numRecords - 1); // 写入 offset，写入的是消息个数
    }
}
```



```

        buffer.putInt(pos - initPos - Records.LOG_OVERHEAD); // 写入 size

        // 写入压缩消息的相关信息, 例如, 时间戳、压缩算法等, MAGIC_VALUE 为 1
        Record.write(buffer, maxTimestamp, null, null, type, 0, -1);

        // 计算并写入整个压缩消息 value 部分的长度
        int valueSize = pos - initPos - Records.LOG_OVERHEAD - Record.RECORD_
OVERHEAD;
        buffer.putInt(initPos + Records.LOG_OVERHEAD + Record.KEY_OFFSET_V1,
            valueSize);

        // 计算并写入 CRC32 校验码
        long crc = Record.computeChecksum(buffer,
            initPos + Records.LOG_OVERHEAD + Record.MAGIC_OFFSET,
            pos - initPos - Records.LOG_OVERHEAD - Record.MAGIC_OFFSET);

        Utils.writeUnsignedInt(buffer, initPos + Records.LOG_OVERHEAD
            + Record.CRC_OFFSET, crc);
        buffer.position(pos); // 还原 buffer 的 position 指针
        // 更新压缩率 (略)
    }
}

```

细心的读者可能会问, 服务端需要为每个消息分配 offset, 要对消息解压缩吗? 这里的设计很巧妙, 原理如下:

(1) 当生产者产生创建压缩信息的时候, 对压缩消息设置的 offset 是内部 offset (inner offset), 即分配给每个消息的 offset 分别是 0、1、2……请读者回顾第 2 章介绍的 `RecordBatch.tryAppend()` 方法。

(2) 在 Kafka 服务端为消息分配 offset 时, 会根据外层消息中记录的内层压缩消息的个数为外层消息分配 offset, 为外层消息分配的 offset 是内层压缩消息中最后一个消息的 offset 值, 如图 4-18 所示。

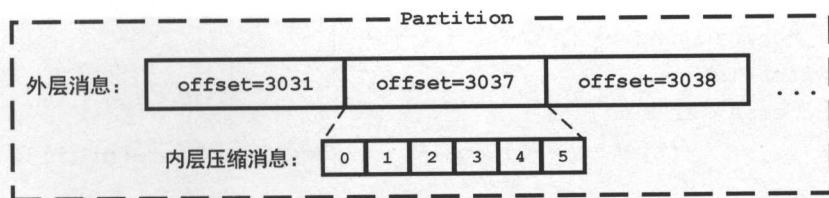


图 4-18

(3) 当消费者获取压缩消息后进行解压缩, 就可以根据内部消息的、相对的 offset 和外层消息的 offset 计算出每个消息的 offset 值了。

迭代压缩消息

上文提到, 消费者获取的压缩消息格式与生产者发送出去的格式应该一模一样。在第3章介绍 `Fetcher.parseFetchData()` 方法解析 `CompleteFetch` 时, 对 `MemoryRecords` 进行迭代。下面来看看 `MemoryRecords` 的迭代器的相关实现, 其中就涉及与压缩消息相关的实现。

`MemoryRecords` 的迭代器是其静态内部类 `RecordsIterator`, 它继承了 `AbstractIterator` 抽象类。`AbstractIterator` 实现了 `Iterator` 接口, 它对 `Iterator` 进行了实现和简化。`AbstractIterator` 只暴露了一个 `makeNext()` 抽象方法供子类实现, 此方法主要负责用于创建下一个迭代项。这样, 开发人员就不用了解 `Iterator` 接口的实现细节了, 只关注如何创建下一迭代项即可。

`AbstractIterator` 中使用 `next` 字段指向迭代的下一项, 使用 `state` 字段标识当前迭代器的状态, `state` 字段是 `State` 枚举类型, 其取值和含义如下所述。

- `READY`: 迭代器已经准备好迭代下一项。
- `NOT_READY`: 迭代器未准备好迭代下一项, 需要调用 `maybeComputeNext()`。
- `DONE`: 当前迭代已经结束。
- `FAILED`: 当前迭代器在迭代过程中出现异常。

`AbstractIterator` 的代码如下:

```
public abstract class AbstractIterator<T> implements Iterator<T> {

    private State state = State.NOT_READY;
    private T next;

    @Override
```

```

    public boolean hasNext() {
        switch (state) {
            case FAILED:
                throw new IllegalStateException("Iterator is in failed
state");
            case DONE: // 迭代已结束, 返回 false
                return false;
            case READY: // next 项已经准备好, 返回 true
                return true;
            // NOT_READY 状态, 需要调用 maybeComputeNext() 方法获取 next 项
            default:
                return maybeComputeNext();
        }
    }

    @Override
    public T next() {
        if (!hasNext())
            throw new NoSuchElementException();
        state = State.NOT_READY; // 将 state 字段重置为 NOT_READY 状态
        if (next == null)
            throw new IllegalStateException("Expected item but none
found.");
        return next; // 返回 next
    }

    ... .. // AbstractIterator 不支持 remove() 操作, 调用 remove() 方法会抛出异常

    // 子类在实现 makeNext() 方法中可以通过调用 allDone() 方法结束整个迭代
    protected T allDone() {
        state = State.DONE;
        return null;
    }

    protected abstract T makeNext(); // 子类实现, 返回迭代的下一个迭代项

    private Boolean maybeComputeNext() {

```

```

        state = State.FAILED; // 若在子类的实现 makeNext() 中抛出异常, 则 state
        会处于此状态
        next = makeNext();
        if (state == State.DONE) {
            return false;
        } else { // 在 makeNext() 方法中完成了 next 项的构造
            state = State.READY;
            return true;
        }
    }
}

```

为了同时能够处理压缩消息和非压缩消息, `MemoryRecords.RecordsIterator` 分为两层迭代, 使用 `shallow` 参数标区分。当 `shallow` 为 `true` 时, 认为消息是非压缩消息, 只迭代当前这一层消息, 为方便描述, 我们称之为“浅层迭代” (shallow iteration); 当 `shallow` 为 `false` 时, 不只迭代当前层消息, 还会创建 `Inner Iterator` (也是 `MemoryRecords.RecordsIterator` 对象) 迭代嵌套的压缩消息, 我们称之为“深层迭代” (deep iteration)。

`MemoryRecords.RecordsIterator` 的关键字段如下所述。

- `buffer`: 指向 `MemoryRecords` 的 `buffer` 字段。其中的消息可以是压缩的, 也可以是非压缩的。
- `stream`: 读取 `buffer` 的输入流。如果迭代压缩消息, 则是对应的解压缩输入流。
- `type`: 压缩类型。
- `shallow`: 标识当前 `RecordsIterator` 是深层迭代器还是浅层迭代器。
- `innerIter`: 迭代压缩消息的 `Inner Iterator`。
- `logEntries`: `ArrayDeque<LogEntry>` 类型集合。 `Inner Iterator` 需要迭代的压缩消息集合。 `Outer Iterator` 的此字段始终为 `null`, 其中 `LogEntry` 中封装了消息及其 `offset`。
- `absoluteBaseOffset`: 在 `Inner Iterator` 迭代压缩消息时使用, 用于记录压缩消息中第一个消息的 `offset`, 并根据此字段计算每个消息的 `offset`。 `Outer Iterator` 的此字段始终为 `-1`。

`MemoryRecords.RecordsIterator` 的构造函数也分为两个: 一个是 `public` 的, 提供给外部的 API, 用于创建 `Outer Iterator`; 另一个是 `private` 的, 用于创建 `Inner Iterator`。


```

public RecordsIterator(ByteBuffer buffer, boolean shallow) {
    this.type = CompressionType.NONE; // 外层消息是非压缩的
    this.buffer = buffer; // 指向MemoryRecords.buffer字段
    this.shallow = shallow; // 标识是否为深层迭代器
    this.stream = new DataInputStream(new ByteBufferInputStream(buffer));
// 输入流
    this.logEntries = null;
    this.absoluteBaseOffset = -1;
}

// inner iterator 的构造函数
private RecordsIterator(LogEntry entry) {
    this.type = entry.record().compressionType(); // 指定内层压缩消息的压缩类型
    this.buffer = entry.record().value();
    this.shallow = true;

    // 创建指定压缩类型的输入流
    this.stream = Compressor.wrapForInput(new ByteBufferInputStream(this.
buffer),
        type, entry.record().magic());
    long wrapperRecordOffset = entry.offset(); // 外层消息的 offset
    if (entry.record().magic() > Record.MAGIC_VALUE_V0) {
        this.logEntries = new ArrayDeque<>();
        long wrapperRecordTimestamp = entry.record().timestamp();
        while (true) { // 在这个循环中, 将内层消息全部解压出来并添加到 logEntries 集合中
            try {
                // 对于内层消息, getNextEntryFromStream() 方法是读取并解压缩消息
                // 对于外层消息或非压缩消息, 则仅仅是读取消息
                LogEntry logEntry = getNextEntryFromStream();
                Record recordWithTimestamp = new Record(logEntry.record().
buffer(),
                    wrapperRecordTimestamp, entry.record().
timestampType());
                // 添加到 logEntries 集合中
                logEntries.add(new LogEntry(logEntry.offset(),
recordWithTimestamp));
            } catch (EOFException e) {

```

```

        break;
        // 其他异常处理 (略)
    }
}
// 计算 absoluteBaseOffset
this.absoluteBaseOffset = wrapperRecordOffset - logEntries.
getLast().offset();
} else {
    this.logEntries = null;
    this.absoluteBaseOffset = -1;
}
}
}

```

makeNext() 方法的相关实现代码如下, 下面会通过一个示例介绍深层迭代的整个过程:

```

protected LogEntry makeNext() {
    if (innerDone()) { // 检测当前深层迭代是否已经完成, 或是深层迭代还未开始
        try {
            LogEntry entry = getNextEntry(); // 获取消息
            if (entry == null) // 获取不到消息, 调用 allDone() 方法结束迭代
                return allDone();

            // 在 Inner Iterator 中计算每个消息的 absoluteOffset
            if (absoluteBaseOffset >= 0) {
                long absoluteOffset = absoluteBaseOffset + entry.offset();
                entry = new LogEntry(absoluteOffset, entry.record());
            }

            // 根据压缩类型和 shallow 参数决定是否创建 Inner Iterator
            CompressionType compression = entry.record().compressionType();
            if (compression == CompressionType.NONE || shallow) {
                return entry;
            } else {
                // 创建 Inner Iterator, 每迭代一个外层消息, 创建一个 Inner Iterator 用迭代其中的内层消息
                innerIter = new RecordsIterator(entry);
                return innerIter.next(); // 迭代内层消息
            }
        }
    }
}

```

```

        }
    } catch (Exception e) {
        // 异常处理 (略)
    }
} else {
    return innerIter.next();
}
}

// 下面是 getNextEntry() 方法的实现
private LogEntry getNextEntry() throws IOException {
    if (logEntries != null)
        return getNextEntryFromEntryList(); // 从 logEntries 队列中获取 LogEntry
    else
        return getNextEntryFromStream(); // 从 buffer 中获取 LogEntry
}

// 下面是 getNextEntryFromStream() 方法的实现
private LogEntry getNextEntryFromStream() throws IOException {
    long offset = stream.readLong(); // 读取 offset
    int size = stream.readInt(); // 读取消息长度
    if (size < 0)
        throw new IllegalStateException("Record with size " + size);

    ByteBuffer rec;
    if (type == CompressionType.NONE) { // 未压缩消息的处理
        rec = buffer.slice(); // rec 是 buffer 的分片
        int newPos = buffer.position() + size;
        if (newPos > buffer.limit())
            return null;
        buffer.position(newPos); // 修改 buffer 的 position
        rec.limit(size);
    } else { // 处理压缩的消息
        byte[] recordBuffer = new byte[size];
        // 从 stream 中读取数据, 此过程会解压消息
        stream.readFully(recordBuffer, 0, size);
        rec = ByteBuffer.wrap(recordBuffer);
    }
}

```



```

    }
    return new LogEntry(offset, new Record(rec));
}

```

图 4-19 展示了两种不同的迭代的过程，为了读者更好地理解整个迭代过程，下面以此图为例进行说明。首先通过 public 构造函数创建 MemoryRecords.RecordsIterator 对象作为浅层迭代器并调用 next() 方法，此时 state 字段为 NOT_READY，调用 makeNext() 方法准备迭代项。在 makeNext() 方法中会判断深层迭代是否完成（即 innerDone() 方法），当前未开始深层迭代则调用 getNextEntryFromStream() 方法获取 offset 为 3031 的消息，如图 4-19 中的步骤①。之后检测 3031 消息的压缩格式，假设采用 GZIP 的压缩格式，则通过通过 private 构造函数创建 MemoryRecords.RecordsIterator 对象作为深层迭代器，在构造过程中会创建对应的解压输入流，然后调用 getNextEntryFromStream() 方法解压 offset 为 3031 的外层消息，其中嵌套的压缩消息形成 logEntries 队列。然后，调用深层迭代器的 next() 方法，因为不存在第三层迭代且 logEntries 不为空，则从 logEntries 集合中获取消息并返回，此过程对应图 4-19 的步骤②。后续迭代中深层迭代未完成，则直接从 logEntries 集合中返回消息，图 4-19 中的步骤③~⑦都会重复此过程。当深层迭代完成后，调用 getNextEntryFromStream() 方法获取 offset 为 3032 的消息，如图 4-19 中的步骤⑧。后续迭代过程与上述过程重复，不再赘述。

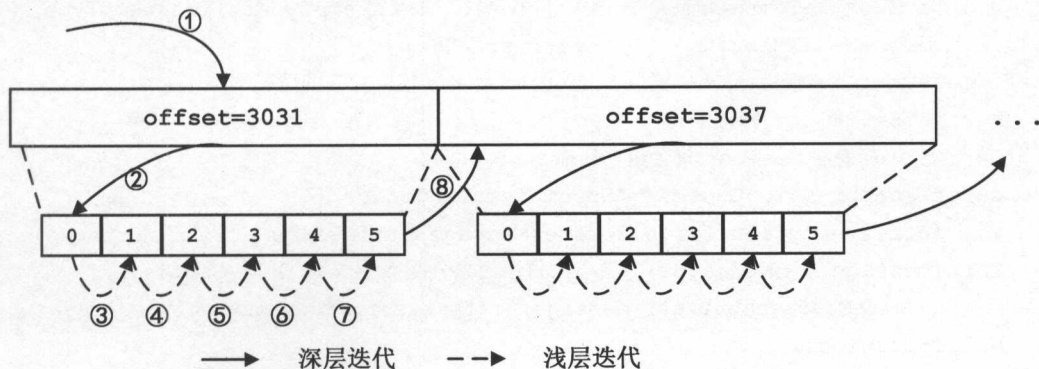


图 4-19

ByteBufferMessageSet 分析

介绍完生产者和消费者对压缩消息的处理过程，我们回到服务端，开始对 ByteBufferMessageSet 的分析，它底层使用 ByteBuffer 保存消息数据。ByteBufferMessageSet 的角色和功能与 MemoryRecords 类似。ByteBufferMessageSet 提供了三个方面的功能：

(1) 将 Message 集合按照指定的压缩类型进行压缩，此功能主要用于构建 ByteBufferMessageSet 对象，通过 ByteBufferMessageSet.create() 方法完成。

提供迭代器，实现深层迭代和浅层迭代两种迭代方式。

(3) 提供了消息验证和 offset 分配的功能。

在 ByteBufferMessageSet.create() 方法中实现了消息的压缩以及 offset 分配，其步骤如下所示。

(1) 如果传入的 Message 集合为空，则返回空 ByteBuffer。

(2) 如果要求不对消息进行压缩，则通过 OffsetAssigner 分配每个消息的 offset，在将消息写入到 ByteBuffer 之后，返回 ByteBuffer。OffsetAssigner 的功能是存储一串 offset 值，并像迭代器那样逐个返回，OffsetAssigner 的实现很简单，读者可参看源码学习。

(3) 如果要求对消息进行压缩，则先将 Message 集合按照指定的压缩方式进行压缩并保存到缓冲区，同时也会完成 offset 的分配，然后按照压缩消息的格式写入外层消息，最后将整个外层消息所在的 ByteBuffer 返回。

ByteBufferMessageSet.create() 方法的代码如下：

```
private def create(offsetAssigner: OffsetAssigner, compressionCodec: CompressionCodec,
    wrapperMessageTimestamp: Option[Long], timestampType: TimestampType,
    messages: Message*): ByteBuffer = {
  if (messages.isEmpty) // 情况 1: 处理 Message 集合为空的情况
    MessageSet.Empty.buffer
  // 情况 2: 不需要对 Message 集合进行压缩
  else if (compressionCodec == NoCompressionCodec) {
    val buffer = ByteBuffer.allocate(MessageSet.messageSetSize(messages))
    for (message <- messages) // 为每个消息分配 offset，并写入 buffer
      writeMessage(buffer, message, offsetAssigner.nextAbsoluteOffset())
    buffer.rewind()
    buffer // 返回 buffer
  } else { // 处理压缩的情况，情况 3
    // 得到 Magic 值和时间戳（略）
    var offset = -1L
    // 底层使用 byte 数组保存写入的压缩数据
    val messageWriter = new MessageWriter(...)
```

```

messageWriter.write(codec = compressionCodec,
    timestamp = magicAndTimestamp.timestamp,
    timestampType = timestampType, magicValue = magicAndTimestamp.
magic) {
    outputStream =>
    // 创建指定压缩类型的输出流
    val output = new DataOutputStream(CompressionFactory(compressionCodec,
        magicAndTimestamp.magic, outputStream))
    for (message <- messages) { // 遍历写入内层压缩消息
        offset = offsetAssigner.nextAbsoluteOffset()
        // Magic 值为 1, 写入的是相对 offset; Magic 值为 0, 写入的是 offset
        if (magicAndTimestamp.magic > Message.MagicValue_V0)
            output.writeLong(offsetAssigner.toInnerOffset(offset))
        else
            output.writeLong(offset)
        output.writeInt(message.size) // 写入 size
        output.write(message.buffer.array, message.buffer.arrayOffset,
            message.buffer.limit) // 写入 Message 数据
    }
}

val buffer = ByteBuffer.allocate(messageWriter.size + MessageSet.
LogOverhead)
// 按照消息格式写入整个外层消息。注意, 外层消息的 offset 是最后一条内层消息的 offset
writeMessage(buffer, messageWriter, offset)
buffer.rewind()
buffer
}
}

```

ByteBufferMessageSet 提供的浅层迭代器和深层迭代器与 MemoryRecords 的迭代器的实现和功能都十分类似, 只是换成使用 Scala 语言实现。由于篇幅限制, 这里就不再赘述了, 相信读者理解了 MemoryRecords. RecordsIterator 的实现后, 能很轻松地看懂 ByteBufferMessageSet 此部分的实现。

ByteBufferMessageSet.validateMessagesAndAssignOffsets() 方法实现了验证消息并分配 offset 的功能, 需要验证的部分如下:

- 检查 Magic Value;
- 检查时间戳与时间戳类型;
- 对于压缩消息需要检查它是否有 key;
- 可以重新设定时间戳类型和时间戳;
- 进行 offset 分配;
- 如果消息压缩类型与 Broker 指定压缩类型不一致, 需要进行重新压缩。

下面是 validateMessagesAndAssignOffsets() 方法的代码, 代码比较长, 略作简化。注意该方法的第一个参数是分配 offset 的起始值, 其他的参数比较好理解, 不再赘述; 该方法的第二个返回值表示 ByteBufferMessageSet 中 Message 集合的长度是否会发生变化。

```
private[kafka] def validateMessagesAndAssignOffsets(offsetCounter: LongRef,
    now: Long, sourceCodec: CompressionCodec, targetCodec: CompressionCodec,
    compactedTopic: Boolean = false,
    messageFormatVersion: Byte = Message.CurrentMagicValue,
    messageTimestampType: TimestampType,
    messageTimestampDiffMaxMs: Long): (ByteBufferMessageSet, Boolean) = {

  if (sourceCodec == NoCompressionCodec && targetCodec == NoCompressionCodec) {
    // 检测所有 Message 的 Magic value 是否与指定的 Magic value 一致
    if (!isMagicValueInAllWrapperMessages(messageFormatVersion)) {-
      // 因为存在 Message 的 Magic value 不一致, 则需要进行统一, 可能导致消息总长度变化,
      // 需要创建新的 ByteBufferMessageSet。同时还会进行 offset 的分配, 验证并更新
      // CRC32、时间戳等信息
      (convertNonCompressedMessages(...), true)
    } else {
      // 处理非压缩消息且 magic 值统一的情况, 由于 Magic 值确定, 长度不会改变。主要是进
      // 行 offset 分配, 验证并更新 CRC32、时间戳等信息
      (validateNonCompressedMessagesAndAssignOffsetInPlace(offsetCounter,
        now,
        compactedTopic, messageTimestampType, messageTimestampDiffMaxMs),
        false)
    }
  } else { // 下面是处理压缩消息的情况
```



```

// inPlaceAssignment 标识是否可以直接复用当前 ByteBufferMessage 对象。下面四种
// 情况不能复用:
// 情况 1. 消息当前压缩类型与此 Broker 指定的压缩类型不一致, 需重新压缩
// 情况 2. Magic 为 0 时, 需要重写消息的 offset 为绝对 offset
// 情况 3. Magic 大于 0, 但内部压缩消息某些字段需要修改, 例如时间戳
// 情况 4. 消息格式需要转换
var inPlaceAssignment = sourceCodec == targetCodec    // 检查情况 1
    && messageFormatVersion > Message.MagicValue_V0 // 检查情况 2
.....
// 遍历内层压缩消息, 此步骤会解压
this.internalIterator(isShallow = false).foreach {
    messageAndOffset => val message = messageAndOffset.message
    validateMessageKey(message, compactedTopic) // 检测消息的 key
    if (message.magic > Message.MagicValue_V0 &&
        messageFormatVersion > Message.MagicValue_V0) {
        .
        // 检测时间戳
        validateTimestamp(message, now,
            messageTimestampType, messageTimestampDiffMaxMs)

        // 检测情况 3, 检查内部 offset 是否正常
        if (messageAndOffset.offset != expectedInnerOffset.getAndIncrement())
            inPlaceAssignment = false
        maxTimestamp = math.max(maxTimestamp, message.timestamp)
    }
    if (message.magic != messageFormatVersion) // 检测情况 4
        inPlaceAssignment = false

    // 保存通过上述检测和转换的 Message 集合
    validatedMessages += message.toFormatVersion(messageFormatVersion)
}

if (!inPlaceAssignment) { // 不能复用当前 ByteBufferMessage 对象的场景
    .....
// 创建新 ByteBufferMessageSet 对象, 重新压缩。此时调用上面介绍的 create() 方法进行压缩
    (new ByteBufferMessageSet(compressionCodec = targetCodec, offsetCounter
=

```



```

        offsetCounter, wrapperMessageTimestamp = wrapperMessageTimestamp,
            timestampType = messageTimestampType, messages =
validatedMessages:*), true)
    } else { // 复用当前 ByteBufferMessageSet 对象, 这样可以减少一次压缩操作

        // 更新外层消息的 offset, 将其 offset 更新为内部最后一条压缩消息的 offset
        buffer.putLong(0, offsetCounter.addAndGet(validatedMessages.size)-1)
        ... ..
        // 更新外层消息的时间戳、attribute 和 CRC32
        buffer.putLong(timestampOffset, now)
        buffer.put(attributeOffset, messageTimestampType.updateAttributes
(attribute))
        Utils.writeUnsignedInt(buffer, ..., wrapperMessage.computeChecksum)
        buffer.rewind()
        (this, false)
    }
}
}
}

```

ByteBufferMessageSet 中的其他方法也是用来辅助实现上述方法的, 感兴趣的读者可以参考相关源码学习。

最后, 我们回到本节开始的那个问题, FileMessageSet.append() 方法会将 ByteBufferMessageSet 中的全部数据追加到日志文件中, 对于压缩消息来说, 多条压缩消息就以一个外层消息的状态存在于分区日志文件中了。当消费者获取消息时也会得到压缩的消息, 从而实现“端到端压缩”。

4.3.4 OffsetIndex

为了提高查找消息的性能, 从 Kafka 0.8 版本开始, 为每个日志文件添加了对应的索引文件。OffsetIndex 对象对应管理磁盘上的一个索引文件, 与上一节分析的 FileMessageSet 共同构成一个 LogSegment 对象。

首先来介绍索引文件中索引项的格式: 每个索引项为 8 字节, 分为两部分, 第一部分是相对 offset, 占 4 个字节; 第二部分是物理地址, 也就是其索引消息在日志文件中对应的 position 位置, 占 4 个字节。这样就实现了 offset 与物理地址之间的映射。相对 offset 表示的是消息相对于 baseOffset 的偏移量。例如, 分段后的一个日志文件的 baseOffset 是 20, 当然, 它的文件名就是 20.log, 那么 offset 为 23 的 Message 在索引文件中的相

对 offset 就是 $23 - 20 = 3$ 。消息的 offset 是 Long 类型，4 个字节可能无法直接存储消息的 offset，所以使用相对 offset，这样可以减小索引文件占用的空间。

Kafka 使用稀疏索引的方式构造消息的索引，它不保证每个消息在索引文件中都有对应的索引项，这算是磁盘空间、内存空间、查找时间等多方面的折中。不断减小索引文件大小的目的是为了将索引文件映射到内存，在 OffsetIndex 中会使用 MappedByteBuffer 将索引文件映射到内存中。

介绍完了索引文件的相关概念后，我们来介绍 OffsetIndex 的字段。

- `_file`: 指向磁盘上的索引文件。
- `baseOffset`: 对应日志文件中第一个消息的 offset。
- `mmap`: 用来操作索引文件的 MappedByteBuffer。
- `lock`: ReentrantLock 对象，在对 mmap 进行操作时，需要加锁保护。
- `_entries`: 当前索引文件中的索引项个数。
- `_maxEntries`: 当前索引文件中最多能够保存的索引项个数。
- `_lastOffset`: 保存最后一个索引项的 offset。

在 OffsetIndex 初始化的过程中会初始化上述字段，因为会有多个 Handler 线程并发写入索引文件，所以这些字段使用 `@volatile` 修饰，保证线程之间的可见性。初始化代码如下：

```
@volatile
private[this] var mmap: MappedByteBuffer = {
  // 如果索引文件不存在，则创建新文件并返回 true，反之返回 false
  val newlyCreated = _file.createNewFile()
  val raf = new RandomAccessFile(_file, "rw")
  try {
    if (newlyCreated) { // 对于新创建的索引文件，进行扩容
      if (maxIndexSize < 8)
        throw new IllegalArgumentException("Invalid max index size: " +
maxIndexSize)
      // 根据 maxIndexSize 的值对索引文件进行扩容，扩容结果是小于 maxIndexSize 的最
      // 大的 8 的倍数
      raf.setLength(roundToExactMultiple(maxIndexSize, 8))
    }
  }
  // 进行内存映射
```

```

val len = raf.length()
val idx = raf.getChannel().map(FileChannel.MapMode.READ_WRITE, 0, len)

if (newlyCreated) // 将新创建的索引文件的 position 设置为 0, 从头开始写文件
    idx.position(0)
else
    // 对于原来就存在的索引文件, 则将 position 移动到所有索引项的结束位置, 防止数据覆盖
    idx.position(roundToExactMultiple(idx.limit, 8))
idx // 返回 MappedByteBuffer
} finally {
    CoreUtils.swallow(raf.close())
}
}

@volatile
private[this] var _entries = mmap.position / 8 // 索引项个数

@volatile
private[this] var _maxEntries = mmap.limit / 8 // 最大索引项个数

@volatile
// 读取最后一个索引项的 offset
private[this] var _lastOffset = readLastEntry.offset

```

OffsetIndex 提供了向索引文件中添加索引项的 `append()` 方法, 将索引文件截断到某个位置的 `truncateTo()` 方法和 `truncateToEntries()` 方法, 进行文件扩容的 `resize()` 方法。这些方法实际上都是通过 `mmap` 字段的相关操作完成的, 这里不再赘述, 请读者参考源码。

OffsetIndex 中最常用的还是查找相关的方法, 使用的是二分查找, 涉及的方法是 `indexSlotFor()` 和 `lookup()`。值得注意的地方是, 查找的目标是小于 `targetOffset` 的最大 `offset` 对应的物理地址 (`position`)。下面是 `lookup()` 方法的代码:

```

def lookup(targetOffset: Long): OffsetPosition = {
    maybeLock(lock) { // Window 操作系统需要加锁, 其他操作系统不需要加锁
        val idx = mmap.duplicate // 创建一个副本
        val slot = indexSlotFor(idx, targetOffset) // 二分查找的具体实现
        if (slot == -1)
            OffsetPosition(baseOffset, 0)
    }
}

```



```

else
    // 将 offset 和物理地址 (position) 封装成 OffsetPosition 对象并返回
    OffsetPosition(baseOffset + relativeOffset(idx, slot), physical(idx,
slot))
    // relativeOffset() 方法和 physical() 方法是获取索引项内容的辅助方法, 分别实现
    // 了读取索引项中的相对 offset 和索引项中的物理地址 (position) 的功能
}
}

private def indexSlotFor(idx: ByteBuffer, targetOffset: Long): Int = {
    val relOffset = targetOffset - baseOffset // 将 offset 转换成相对 offset
    // 某些异常情况检查, 例如, 没有索引项或 targetOffset 小于 baseOffset (略)
    // 下面是标准的二分查找算法的实现, 不多赘述了
    var lo = 0
    var hi = _entries - 1
    while (lo < hi) {
        val mid = ceil(hi / 2.0 + lo / 2.0).toInt
        val found = relativeOffset(idx, mid) // 获取相对 offset
        if (found == relOffset)
            return mid
        else if (found < relOffset)
            lo = mid
        else
            hi = mid - 1
    }
    lo // 注意: 如果找不到的 targetOffset 对应的索引项, 则返回小于 targetOffset 的、最
    // 大的索引项位置
}

```

OffsetIndexIndex 的实现就介绍到这里。下一小节来分析 LogSegment 这个类。

4.3.5 LogSegment

为了防止 Log 文件过大, 将 Log 切分成多个日志文件, 每个日志文件对应一个 LogSegment。在 LogSegment 中封装了一个 FileMessageSet 和一个 OffsetIndex 对象, 提供日志文件和索引文件的读写功能以及其他辅助功能。

下面先来看 LogSegment 的核心字段。

- **log**: 用于操作对应日志文件的 `FileMessageSet` 对象。
- **index**: 用于操作对应索引文件的 `OffsetIndex` 对象。
- **baseOffset**: `LogSegment` 中第一条消息的 `offset` 值。
- **indexIntervalBytes**: 索引项之间间隔的最小字节数。
- **bytesSinceLastIndexEntry**: 记录自从上次添加索引项之后, 在日志文件中累计加入的 `Message` 集合的字节数, 用于判断下次索引项添加的时机。
- **created**: 标识 `LogSegment` 对象创建时间, 当调用 `truncateTo()` 方法将整个日志文件清空时, 会将此字段重置为当前时间。参与创建新 `LogSegment` 的条件判断, 在介绍 `Log` 类时会详细介绍。

在 `LogSegment.append()` 方法中实现了追加消息的功能, 可能有多个 `Handler` 线程并发写入同一个 `LogSegment`, 所以调用此方法时必须保证线程安全, 在后面分析 `Log` 类时会看到相应的同步代码。另外, 注意 `append()` 方法的参数, 其第二个参数 `messages` 表示的是待追加的消息集合, 第一个参数 `offset` 表示 `messages` 中的第一条消息的 `offset`, 如果是压缩消息, 则是第一条内层消息的 `offset`, 如图 4-20 所示。`append()` 方法的代码如下:

```
def append(offset: Long, messages: ByteBufferMessageSet) {
  if (messages.sizeInBytes > 0) {
    // 检测是否在满足添加索引项的条件
    if (bytesSinceLastIndexEntry > indexIntervalBytes) {
      index.append(offset, log.sizeInBytes()) // 添加索引
      // 成功添加索引后, bytesSinceLastIndexEntry 重置为 0
      this.bytesSinceLastIndexEntry = 0
    }

    log.append(messages) // 写日志文件
    // 更新 bytesSinceLastIndexEntry
    this.bytesSinceLastIndexEntry += messages.sizeInBytes
  }
}
```



图 4-20

读取消息的功能由 `LogSegment.read()` 方法实现，它有四个参数。

- `startOffset`: 指定读取的起始消息的 `offset`。
- `maxOffset`: 指定读取结束的 `offset`，可以为空。
- `maxSize`: 指定读取的最大字节数。
- `maxPosition`: 指定读取的最大物理地址，可选参数，默认值是日志文件的大小。

在读取日志文件之前，需要将 `startOffset` 和 `maxOffset` 转化为对应的物理地址才能使用。这个转换在 `translateOffset()` 方法中实现，我们先通过一个例子来介绍其功能。现假设 `startOffset` 是 1017，图 4-21 展示了将 1017 这个 `offset` 转换成对应的物理地址的过程。

(1) 我们将 `absoluteOffset` 转换成 Index File 中使用的相对 `offset`，得到 17。通过 `OffsetIndex.lookup()` 方法查找 Index File，得到 (7,700) 这个索引项，如图 4-21 中步骤①所示。

(2) 根据 (7,700) 索引项，我们从 MessageSet File 中 `position=700` 处开始查找 `absoluteOffset` 为 1017 的消息，如图 4-21 中步骤②所示。

(3) 通过 `FileMessageSet.searchFor()` 方法遍历查找 `FileMessageSet`，得到 (1018,800) 这个位置信息，如图 4-21 中步骤③所示。

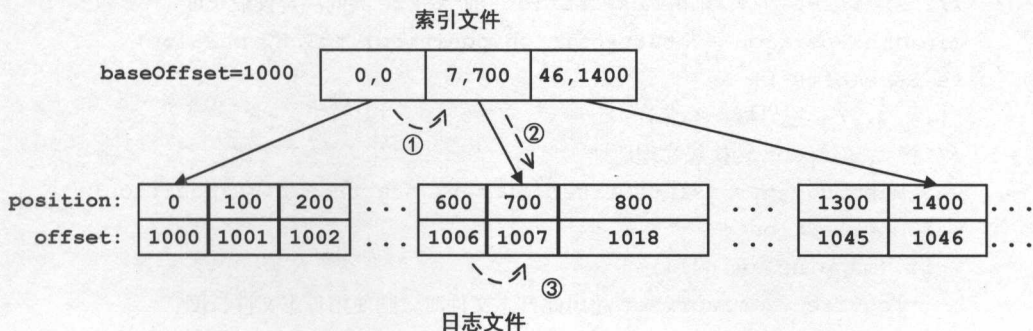


图 4-21

读者可能会问，我们的目标 offset 是 1017，为什么会返回 1018 呢？在这个示例中，offset=1017 的消息与其他的消息被压缩后一起构成了 offset=1018 这条外层消息，并存入了日志文件中。translateOffset() 方法的代码如下：

```
private[log] def translateOffset(offset: Long, startingFilePosition: Int = 0):
  OffsetPosition = {
    // 通过 OffsetIndex 完成第 1 步对索引文件的查找
    val mapping = index.lookup(offset)
    // 通过 FileMessageSet 完成第 2、3 步对日志文件的查找
    log.searchFor(offset, max(mapping.position, startingFilePosition))
  }
```

了解了 offset 与物理地址之间的转换后，再来看 read() 方法就比较简单了。注意读取的结束位置由 maxOffset、maxSize、maxPosition 共同决定。LogSegment.read() 方法的代码如下，其中省略了一些边界检查的代码：

```
def read(startOffset: Long, maxOffset: Option[Long], maxSize: Int,
        maxPosition: Long = size): FetchDataInfo = {
  val logSize = log.sizeInBytes // 日志文件的长度
  // 将 startOffset 转换为物理地址
  val startPosition = translateOffset(startOffset)
  ... .. // 边界检查 (略)
  val offsetMetadata = new LogOffsetMetadata(startOffset, this.baseOffset,
    startPosition.position)

  val length = maxOffset match { // 计算读取的字节数
    case None =>
      // maxOffset 为空，则由 maxPosition、maxSize 共同决定读取长度
      min((maxPosition - startPosition.position).toInt, maxSize)
    case Some(offset) =>
      ... .. // 边界检查 (略)
      // 将 maxOffset 转换成物理地址
      val mapping = translateOffset(offset, startPosition.position)
      val endPosition =
        if (mapping == null)
          logSize // maxOffset 超出此日志文件时，则使用日志文件长度
        else
```

```

        mapping.position

        // 由 maxOffset、maxPosition、maxSize 共同决定读取长度
        min(min(maxPosition, endPosition) - startPosition.position, maxSize).
toInt
    }
    // maxOffset 通常是 Replica 的 HW，即消费者最多只能读取到 hw 这个位置的消息

    // 通过对前面对 FileMessageSet.read() 方法的介绍，FetchDataInfo 的第二个参数是按
    // 照读取起始位置和长度生成一个分片的 FileMessageSet 对象，并没有真正读取数据到内存中
    FetchDataInfo(offsetMetadata, log.read(startPosition.position, length))
}

```

LogSegment 中还有一个值得注意的方法是 `recover()` 方法，其主要功能是根据日志文件重建索引文件，同时验证日志文件中消息的合法性。在重建索引文件过程中，如果遇到了压缩消息需要进行解压，主要原因是因为索引项中保存的相对 `offset` 是第一条消息的 `offset`，而外层消息的 `offset` 是压缩消息集合中的最后一条消息的 `offset`。`recover()` 方法的代码如下：

```

def recover(maxMessageSize: Int): Int = {
    // 清空索引文件，底层只是移动 position 指针，后续的写入会覆盖原有的内容
    index.truncate()
    index.resize(index.maxIndexSize) // 修改索引文件大小
    var validBytes = 0 // validBytes 记录了已经通过验证的字节数
    var lastIndexEntry = 0 // 最后一个索引项对应的物理地址
    val iter = log.iterator(maxMessageSize) // FileMessageSet 的迭代器
    try {
        while (iter.hasNext) {
            val entry = iter.next
            entry.message.ensureValid() // 验证 Message 是否合法，验证失败就抛异常
            // 符合添加索引项的条件
            if (validBytes - lastIndexEntry > indexIntervalBytes) {
                val startOffset =
                    entry.message.compressionCodec match {
                        case NoCompressionCodec =>
                            entry.offset

```

也是当前副本的 LEO (LogEndOffset)，LEO 的初始值为 0，LEO 的初始值为 0，LEO 的初始值为 0。


```

        case _ => // 对压缩消息解压缩, 获取其第一个消息的 offset
            ByteBufferMessageSet.deepIterator(entry).next().offset
    }
    index.append(startOffset, validBytes) // 添加索引项
    lastIndexEntry = validBytes // 修改 lastIndexEntry
}
validBytes += MessageSet.entrySize(entry.message) // 累加 validBytes
}
} catch {
    ... ..// 异常处理 (略)
}
val truncated = log.sizeInBytes - validBytes
log.truncateTo(validBytes) // 对日志文件进行截断, 抛弃后面验证失败的 Message
index.trimToValidSize() // 对索引文件进行相应截断
truncated // 返回截掉的字节数
}

```

4.3.6 Log

Log 是对多个 LogSegment 对象的顺序组合, 形成一个逻辑的日志。为了实现快速定位 LogSegment, Log 使用跳表 (SkipList) 对 LogSegment 进行管理。

```

private val segments: ConcurrentNavigableMap[java.lang.Long, LogSegment] =
    new ConcurrentSkipListMap[java.lang.Long, LogSegment]

```

跳表是一种随机化的数据结构, 它的查找效率和红黑树差不多, 但是插入 / 删除操作却比红黑树简单很多。目前在 Redis 等开源软件中都能看到它的身影, 在 JDK 中也提供了跳表的实现——ConcurrentSkipListMap, 而且 ConcurrentSkipListMap 还是一个线程安全的实现, 有兴趣的读者可以参考其源码。

在 Log 中, 将每个 LogSegment 的 baseOffset 作为 key, LogSegment 对象作为 value, 放入 segments 这个跳表中管理, 如图 4-22 所示。

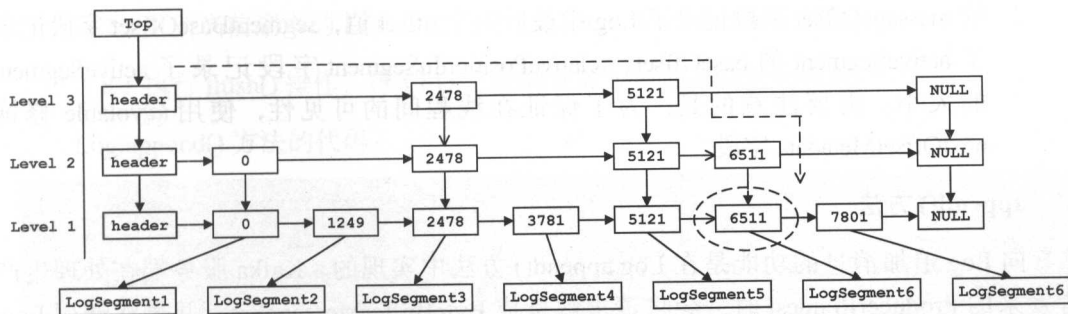


图 4-22

例如，我们现在要查找 offset 大于 6570 的消息，可以首先通过 segments 快速定位到消息所在的 LogSegment 对象，定位过程如图 4-22 中的虚线所示。之后使用前面介绍的 LogSegment.read() 方法，先按照 OffsetIndex 进行索引，然后从日志文件中进行读取。

向 Log 中追加消息时是顺序写入的，那么只有最后一个 LogSegment 能够进行写入操作，在其之前的所有 LogSegment 都不能写入数据。最后一个 LogSegment 使用 Log.activeSegment() 方法获取，即 segments 集合中最后一个元素，为了描述方便，我们将此 Segment 对象称为“activeSegment”。随着数据的不断写入，当 activeSegment 的日志文件大小到达一定阈值时，就需要创建新的 activeSegment，之后追加的消息将写入新的 activeSegment。

介绍完了 Log 的基本原理后，来看一下 Log 类中的关键字段。

- dir: Log 对应的磁盘目录，此目录下存放了每个 LogSegment 对应的日志文件和索引文件。
- lock: 可能存在多个 Handler 线程并发向同一个 Log 追加消息，所以对 Log 的修改操作需要进行同步。
- segments: 用于管理 LogSegment 集合的跳表。
- config: Log 相关的配置信息，具体配置项在具体代码中分析。
- recoveryPoint: 指定恢复操作的起始 offset，recoveryPoint 之前的 Message 已经刷新到磁盘上持久存储，而其后的消息则不一定，出现宕机时可能会丢失。所以只需要恢复 recoveryPoint 之后的消息即可。
- nextOffsetMetadata: LogOffsetMetadata 对象。主要用于产生分配给消息的 offset，同时也是当前副本的 LEO (LogEndOffset)。LEO 的相关介绍请读者参考第 1 章。它

的 `messageOffset` 字段记录了 Log 中最后一个 `offset` 值, `segmentBaseOffset` 字段记录了 `activeSegment` 的 `baseOffset`, `relativePositionInSegment` 字段记录了 `activeSegment` 的大小。需要注意的是, 为了保证在线程间的可见性, 使用 `@volatile` 修饰 `nextOffsetMetadata` 字段。

append() 方法

向 Log 追加消息的功能是在 `Log.append()` 方法中实现的。Kafka 服务端在处理生产者发来的 `ProducerRequest` 时, 会将请求解析成 `ByteBufferMessageSet`, 并最终调用 `Log.append()` 方法完成追加消息, 图 4-23 展示了这一调用关系。

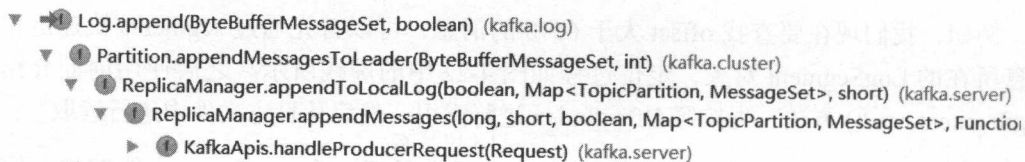


图 4-23

`Log.append()` 方法的大致流程如下:

(1) 首先调用 `Log.analyzeAndValidateMessageSet()` 方法, 对 `ByteBufferMessageSet` 中的 `Message` 数据进行验证, 并返回 `LogAppendInfo` 对象。在 `LogAppendInfo` 中封装了 `ByteBufferMessageSet` 中第一个消息的 `offset`、最后一个消息的 `offset`、生产者采用的压缩方式、追加到 Log 的时间戳、服务端用的压缩方式、外层消息的个数、通过验证的总字节数等信息。

(2) 调用 `Log.trimInvalidBytes()` 方法, 清除未验证通过的 `Message`。

(3) 调用 `ByteBufferMessageSet.validateMessagesAndAssignOffsets()` 方法, 进行内部压缩消息做进一步验证、消息格式转换、调整 `Magic` 值、修改时间戳等操作, 并为 `Message` 分配 `offset`。在 `ByteBufferMessageSet` 小节介绍过了, 这里就不再赘述了。

(4) 如果在 `validateMessagesAndAssignOffsets()` 方法中修改了 `ByteBufferMessageSet` 的长度, 则重新验证 `Message` 的长度是否合法。

(5) 调用 `Log.maybeRoll()` 方法获取 `activeSegment`, 此过程可能分配新的 `activeSegment`。

(6) 将 `ByteBufferMessageSet` 中的消息追加到 `activeSegment` 中, 通过调用 `LogSegment.append()` 方法的实现。

(7) 更新当前副本的 LEO，也就是 `Log.nextOffsetMetadata` 字段。

(8) 执行 `flush()` 操作，将 LEO 之前的全部 Message 刷新到磁盘。

`Log.append()` 方法的代码：

```
def append(messages: ByteBufferMessageSet, assignOffsets: Boolean = true)
    : LogAppendInfo = {
    // 步骤 1: 检测消息长度和 CRC32 校验码, 返回 LogAppendInfo 对象
    val appendInfo = analyzeAndValidateMessageSet(messages)

    // 边界检查: 检查外层消息个数, 如果为 0, 则直接返回 (略)
    // 步骤 2: 将未通过 analyzeAndValidateMessageSet() 方法检查的部分截断
    var validMessages = trimInvalidBytes(messages, appendInfo)
    try {
        lock synchronized { // 加锁
            if (assignOffsets) { // 判断是否需要分配 offset, 默认是需要的

                // 获取 nextOffsetMetadata 记录的 messageOffset 字段, 从此值开始向后分配 offset
                val offset = new LongRef(nextOffsetMetadata.messageOffset)

                // 使用 firstOffset 字段记录第一条消息的 offset, 并不受压缩消息的影响。
                appendInfo.firstOffset = offset.value
                val now = time.milliseconds
                val (validatedMessages, messageSizesMaybeChanged) = try {
                    // 步骤 3: 进一步验证, 并分配 offset。参考 ByteBufferMessageSet 的介绍
                    validMessages.validateMessagesAndAssignOffsets(...)
                } catch {
                    // 异常处理 (略)
                }
            }
            validMessages = validatedMessages
            // lastOffset 记录最后一条消息的 offset, 并不受压缩消息的影响
            appendInfo.lastOffset = offset.value - 1
            if (config.messageTimestampType == TimestampType.LOG_APPEND_TIME)
                appendInfo.timestamp = now // 修改时间戳

            // 步骤 4: 若在 validateMessagesAndAssignOffsets() 方法中修改了
            // ByteBufferMessageSet 的长度, 则需要重新检测消息长度
            // 原理同 analyzeAndValidateMessageSet() 方法 (略)
        }
    }
```



```

    } else {
        // 边界检查 (略)
    }
    // 边界检查: 主要检测待写入的 ByteBufferMessageSet 大小是否大于 LogConfig
    // 配置的 Segment Size (略)

    // 步骤 5: 获取 activeSegment
    val segment = maybeRoll(validMessages.sizeInBytes)
    segment.append(appendInfo.firstOffset, validMessages) // 步骤 6: 追加消息
    updateLogEndOffset(appendInfo.lastOffset + 1) // 步骤 7: 更新 LEO

    // 步骤 8: 检测未刷新到磁盘的数据是否达到一定阈值, 如果是则调用 flush() 方法刷新
    if (unflushedMessages >= config.flushInterval)
        flush()
    appendInfo
}
} catch {
    // 异常处理 (略)
}
}

```

介绍了 `append()` 方法的骨架代码, 下面具体分析其中每个方法的具体实现。步骤 1 中的 `Log.analyzeAndValidateMessageSet()` 方法主要功能是验证消息的长度、CRC32 校验码、内部 `offset` 是否单调递增, 这些验证都是对外层消息进行的, 并不会解压内部的压缩消息。在 `append()` 方法的代码中我们也可以看到, 如果需要进行 `offset` 分配, `analyzeAndValidateMessageSet()` 方法返回的 `LogAppendInfo` 对象记录中的 `firstOffset`、`lastOffset` 甚至时间戳都会被修改。`Log.analyzeAndValidateMessageSet()` 方法的代码如下:

```

private def analyzeAndValidateMessageSet(messages: ByteBufferMessageSet)
    : LogAppendInfo = {
    var shallowMessageCount = 0 // 记录外层消息的数量
    var validBytesCount = 0 // 记录通过验证的 Message 的字节数之和
    var firstOffset, lastOffset = -1L // 记录第一条消息和最后一条消息的 offset
    var sourceCodec: CompressionCodec = NoCompressionCodec
    var monotonic = true // 标识生产者是否为消息分配的内部 offset 是否单调递增
    // 使用浅层迭代器进行迭代, 如果是压缩消息, 并不会解压

```

```

for (messageAndOffset <- messages.shallowIterator) {
    // 记录第一条消息的 offset，此时的 offset 还是生产者分配的 offset
    if (firstOffset < 0)
        firstOffset = messageAndOffset.offset
    if (lastOffset >= messageAndOffset.offset) // 判断内部 offset 是否单调递增
        monotonic = false
    lastOffset = messageAndOffset.offset // 记录最后一条消息的 offset

    val m = messageAndOffset.message
    val messageSize = MessageSet.entrySize(m)
    if (messageSize > config.maxMessageSize) { // 检测 Message 长度
        // 抛出异常（略）
    }
    m.ensureValid() // 检测 Message 的 CRC32 校验码
    shallowMessageCount += 1 // 增加通过检测的外层消息数
    validBytesCount += messageSize // 增加通过检测的字节数
    val messageCodec = m.compressionCodec
    if (messageCodec != NoCompressionCodec)
        sourceCodec = messageCodec // 记录生产者采用的压缩方式
}

val targetCodec = BrokerCompressionCodec.getTargetCompressionCodec(
    config.compressionType, sourceCodec) // 记录服务端采用的压缩方式

LogAppendInfo(firstOffset, lastOffset, Message.NoTimestamp, sourceCodec,
    targetCodec, shallowMessageCount, validBytesCount, monotonic)
}

```

步骤 2 中的 `Log.trimInvalidBytes()` 方法会根据 `analyzeAndValidateMessageSet()` 方法返回的 `LogAppendInfo` 对象，将未通过验证的消息截断。原理如图 4-24 所示，代码比较简单，不再贴出来了。

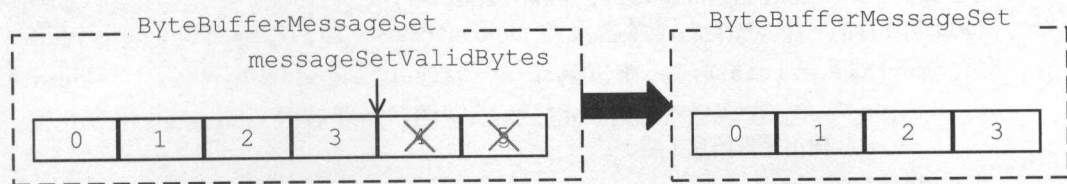


图 4-24

步骤 3 调用的 `ByteBufferMessageSet.validateMessagesAndAssignOffsets()` 方法前面已经介绍过了，不再赘述。步骤 5 中调用的 `Log.maybeRoll()` 方法会检测是否满足创建新 `activeSegment` 的条件，如果满足则创建新 `activeSegment`，然后返回当前的 `activeSegment`。创建新 `activeSegment` 的条件有下面几个：

- 当前 `activeSegment` 的日志大小加上本次待追加的消息集合大小，超过配置的 `LogSegment` 的最大长度。
- 当前 `activeSegment` 的寿命超过了配置的 `LogSegment` 最长存活时间。
- 索引文件满了。

`maybeRoll()` 方法的代码如下，其中创建新 `activeSegment` 的逻辑在 `roll()` 方法中实现：

```
private def maybeRoll(messagesSize: Int): LogSegment = {
  val segment = activeSegment
  if (segment.size > config.segmentSize - messagesSize // 条件 1
    || segment.size > 0 && time.milliseconds - segment.created
      > config.segmentMs - segment.rollJitterMs // 条件 2
      || segment.index.isFull // 条件 3 ) {
    ... .. // 打印 Debug 日志 (略)
    roll() // 创建新的 activeSegment
  } else {
    segment // 不需要创建新的 activeSegment, 直接返回当前的 activeSegment
  }
}

// 下面是 roll() 方法的实现
def roll(): LogSegment = {
  val start = time.nanoseconds
  lock synchronized { // 加锁
    val newOffset = logEndOffset // 获取 LEO
    // 新日志文件的文件名是 [LEO].log
    val logFile = logFilename(dir, newOffset)
    // 新索引文件的文件名是 [LEO].index
    val indexFile = indexFilename(dir, newOffset)
    for (file <- List(logFile, indexFile); if file.exists) {
      ... .. // 输出警告信息 (略)
      file.delete()
    }
  }
}
```



```

segments.lastEntry() match { // segments 的最后一个, 也就是旧 activeSegment
  case null =>
  case entry => {
    // 对索引文件和日志文件都进行截断, 保证文件中只保存了有效字节, 这对预分配的文件尤其重要
    entry.getValue.index.trimToValidSize()
    entry.getValue.log.trim()
  }
}
// 新创建的 LogSegment
val segment = new LogSegment(dir, startOffset = newOffset,
  indexIntervalBytes = config.indexInterval, maxIndexSize = config.
maxIndexSize,
  rollJitterMs = config.randomSegmentJitter, time = time,
  fileAlreadyExists = false, initFileSize = initFileSize,
  preallocate = config.preallocate)

// 将新创建的 Segment 添加到 segments 这个跳表中
val prev = addSegment(segment)
if (prev != null)
  // 之前就存在对应 baseOffset 的 LogSegment, 抛出异常
  throw new KafkaException("...")

// 更新 nextOffsetMetadata, 这次更新的目的是为了更新其中记录的
// activeSegment.baseOffset 和 activeSegment.size, 而 LEO 并不会改变
updateLogEndOffset(nextOffsetMetadata.messageOffset)
// 执行 flush() 操作
scheduler.schedule("flush-log", () => flush(newOffset), delay = 0L)
segment // 返回新建的 activeSegment
}
}

```

在 `Log.roll()` 方法最后使用 `KafkaScheduler` 线程池执行 `flush()` 方法, 它是在 `JDK` 提供的 `ScheduledThreadPoolExecutor` 之上进行的封装和配置。在 `Kafka` 服务端有一部分定时任务是交由 `KafkaScheduler` 线程池执行的。`KafkaScheduler` 线程的实现如下:


```

class KafkaScheduler(val threads: Int,
                     val threadNamePrefix: String = "kafka-scheduler-",
                     daemon: Boolean = true) extends Scheduler with
Logging {

  // JDK 提供的定时任务线程池实现
  private var executor: ScheduledThreadPoolExecutor = null
  private val schedulerThreadId = new AtomicInteger(0)

  override def startup() {
    this synchronized {
      if(isStarted)
        throw new IllegalStateException("This scheduler has already been
started!")
      executor = new ScheduledThreadPoolExecutor(threads) // 配置线程数
      executor.setContinueExistingPeriodicTasksAfterShutdownPolicy(false)
      executor.setExecuteExistingDelayedTasksAfterShutdownPolicy(false)
      executor.setThreadFactory(new ThreadFactory() {
        def newThread(runnable: Runnable):
          Thread = Utils.newThread(
            threadNamePrefix+schedulerThreadId.getAndIncrement(),
            runnable, daemon)) // 默认产生的都是守护线程
      }
    }
  }

  override def shutdown() {
    val cachedExecutor = this.executor
    if (cachedExecutor != null) {
      this synchronized {
        cachedExecutor.shutdown()
        this.executor = null
      }
      cachedExecutor.awaitTermination(1, TimeUnit.DAYS) // 硬编码, 等待一天
    }
  }

  def schedule(name: String, fun: ()=>Unit, delay: Long, period: Long,
unit: TimeUnit)={

```

```

this synchronized {
    val runnable = CoreUtils.runnable { // 将 fun 封装成 Runnable
        try {
            fun() // 调用 fun()
        } catch {
            // 异常处理
        }
    }
    if(period >= 0) // 处理周期性定时任务
        executor.scheduleAtFixedRate(runnable, delay, period, unit)
    else // 处理非周期性定时任务
        executor.schedule(runnable, delay, unit)
}
}
}

```

回到 `append()` 方法继续分析，步骤 6 调用了 `LogSegment.append()`，请参考 `LogSegment` 小节介绍。

最后来看 `flush()` 方法的原理，如图 4-25 所示，`flush()` 方法会将 `recoverPoint`~`LEO` 之间的消息数据刷新到磁盘上，并修改 `recoverPoint` 值。

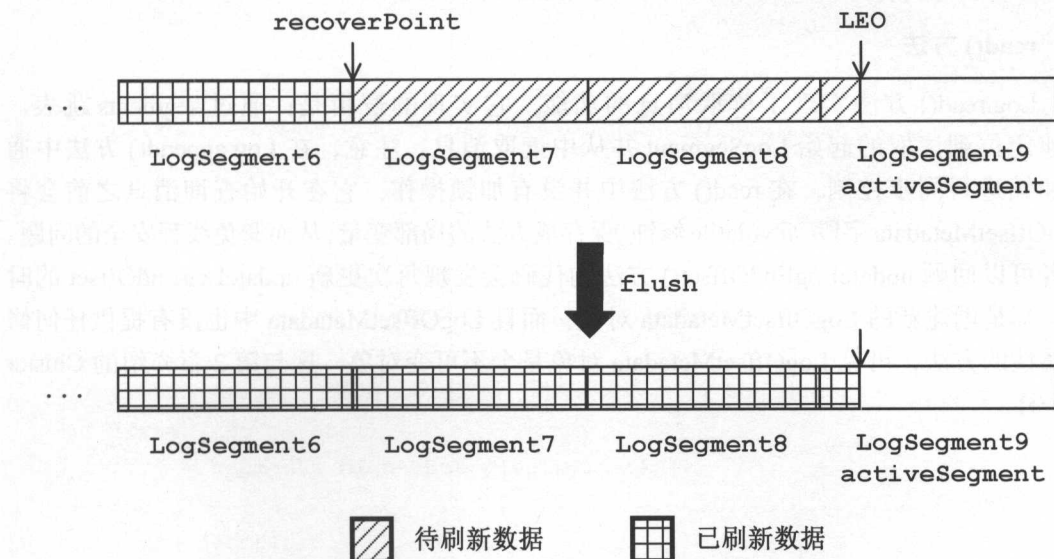


图 4-25

flush() 方法的代码如下:

```
def flush(offset: Long): Unit = {
  // offset 之前的消息已经全部刷新到磁盘, 所以不需要刷新
  if (offset <= this.recoveryPoint)
    return
  // logSegments() 方法, 通过推 segments 这个跳表的操作, 查找到 recoverPoint 和
  // offset 之间的 LogSegment 对象
  for (segment <- logSegments(this.recoveryPoint, offset)) {
    // 调用 LogSegment.flush() 方法会调用日志文件和索引文件的 flush() 方法, 最终调用操
    // 作系统的 fsync 命令刷新磁盘, 保证数据持久性
    segment.flush()
  }
  lock synchronized {
    if (offset > this.recoveryPoint) {
      this.recoveryPoint = offset // 后移 recoveryPoint
      lastflushedTime.set(time.milliseconds) // 修改 lastflushedTime
    }
  }
}
```

整个 Log.append() 方法的功能和实现到这里就分析完了。

read() 方法

Log.read() 方法实现了读取消息的功能, 它实现的逻辑是: 通过 segments 跳表, 快速定位到读取的起始 LogSegment 并从中读取消息。注意, 在 Log.append() 方法中通过加锁进行同步控制, 在 read() 方法中并没有加锁操作, 它在开始查询消息之前会将 nextOffsetMetadata 字段(@volatile 修饰)保存成方法的局部变量, 从而避免线程安全的问题。读者可以回顾 updateLogEndOffset() 方法的代码会发现每次更新 updateLogEndOffset 的时候, 都是创建新的 LogOffsetMetadata 对象, 而且 LogOffsetMetadata 中也没有提供任何修改属性的方法, 可见 LogOffsetMetadata 对象是个不可变对象, 这与第 2 章介绍的 Cluster 类一样。


```

def read(startOffset: Long, maxLength: Int, maxOffset: Option[Long] =
None)
    :FetchDataInfo = {
    // 将 nextOffsetMetadata 保存成局部变量
    val currentNextOffsetMetadata = nextOffsetMetadata
    val next = currentNextOffsetMetadata.messageOffset
    if (startOffset == next) // 边界检测
        return FetchDataInfo(currentNextOffsetMetadata, MessageSet.Empty)

    // 查找 baseOffset 小于 startOffset 且 baseOffset 最大的 LogSegment
    var entry = segments.floorEntry(startOffset)
    if (startOffset > next || entry == null) // 边界检查
        throw new OffsetOutOfRangeException(...)

    while (entry != null) {
        val maxPosition = {
            if (entry == segments.lastEntry) { // 处理读取 activeSegment 的情况——(1)
                val exposedPos = nextOffsetMetadata.relativePositionInSegment.
toLong
                if (entry != segments.lastEntry)
                    // 写线程并发进行了 roll() 操作, 变成了读取非 activeSegment 的场景
                    entry.getValue.size
                else
                    exposedPos
            } else { // 读取的是非 activeSegment 的情况, 直接可以读取到 LogSegment 结尾
                entry.getValue.size
            }
        }
        // 调用 LogSegment.read() 方法读取消息
        val fetchInfo = entry.getValue.read(startOffset, maxOffset, maxLength,
maxPosition)
        // 在此 LogSegment 中没有读取到数据, 则继续读取下一个 LogSegment
        if (fetchInfo == null) {
            entry = segments.higherEntry(entry.getKey)
        } else {
            return fetchInfo
        }
    }
}

```



```
// 查找不到 startOffset 之后的消息
FetchDataInfo(nextOffsetMetadata, MessageSet.Empty)
}
```

这里着重介绍(1)处的代码,为什么需要针对 activeSegment 的读取做特殊的处理呢?在 Kafka 的 Bug 列表中的 [KAFKA-2477] 描述了此 Bug,下面介绍造成这个问题的主要原因。在 Kafka 0.8 版本中未修复这个问题,前面(1)处对应的代码如下:

```
while(entry != null) {
    // 注意这里并没有指定 maxPosition, 在 LogSegment.read() 方法中仅通过 maxOffset 和
    // maxLength 决定读取的长度,这就导致了图 4-26 (a) 的场景
    val messages = entry.getValue.read(startOffset, maxOffset, maxLength)
    if(messages == null)
        entry = segments.higherEntry(entry.getKey) // 如果没有读到消息,继续读取下一个 LogSegment
    else
        return messages
}
```

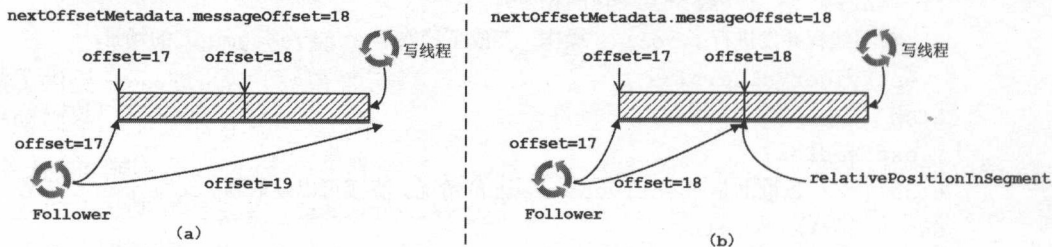


图 4-26

简单描述这种场景:

(1) 前面分析了 append() 方法的逻辑,在写线程调用 append() 方法时,会加锁写入,不会出现多个写线程的并发。现在按照 append() 方法的执行顺序将其分为两个操作:一是先分配 offset 并将 Message 追加到日志文件中,二是更新 nextOffsetMetadata。

现在假设写线程在执行完第一步写入 offset 为 18 的消息后, CPU 时间片到期,线程挂起,导致未对 nextOffsetMetadata.messageOffset 进行更新。

(2) 其他的 Follower 副本发来 Fetch 请求,读取 offset 为 17 以及之后的 Message。

按照 Kafka 0.8 版本的代码，Leader 副本会将 offset 为 17、18 的 Message 全部返回给 Follower。

(3) Follower 处理完 offset 为 17、18 两条 Message，会继续请求 offset 为 19 的 Message，此时，请求的 `startOffset > nextOffsetMetadata.messageOffset`，Follower 就会得到 `OffsetOutOfRangeException`。Follower 认为自己的 Log 出现了问题，会将此 Log 全部删除，并请求从 Leader 重新同步一份过来。

(4) 之后，写线程重新执行，更新 `nextOffsetMetadata`。

在海量数据的情况下，Kafka 中的每个 Log 都很大的（在笔者的实践场景中，一个 Log 大约有 15GB 左右），如果多个 Follower 出现上述重新同步整个 Log 的情况，Leader 副本所在的服务器的 I/O 很快就会被占满，整个服务器都变得不可用。为了处理这个问题，就有了我们看到的对 `activeSegment` 的特殊处理，依然是上述场景，由于 `nextOffsetMetadata` 未更新，`nextOffsetMetadata.relativePositionInSegment` 依然指向 offset 为 17 的 Message 的尾部，限制了 Leader 返回给 Follower 的消息。当 Follower 请求 offset 为 18 的消息时，返回的是消息集合是空，如图 4-26 (b) 所示。

Log 类的其他方法，在下一节介绍 `LogManager` 时还会提到，这里不再单独分析了。

4.3.7 LogManager

在一个 Broker 上的所有 Log 都是由 `LogManager` 进行管理的。`LogManager` 提供了加载 Log、创建 Log 集合、删除 Log 集合、查询 Log 集合等功能，并且启动了 3 个周期性的后台任务以及 Cleaner 线程（可能不止一个线程），分别是：`log-flusher`（日志刷写）任务、`log-retention`（日志保留）任务、`recovery-point-checkpoint`（检查点刷新）任务以及 Cleaner 线程（日志清理）。

下面介绍 `LogManager` 中各个字段的功能。

- `logDirs`: log 目录集合，在 `server.properties` 配置文件中通过 `log.dirs` 项指定的多个目录。每个 log 目录下可以创建多个 Log，每个 Log 都有自己对应的目录，不要混淆。`LogManager` 在创建 Log 时会选择 Log 最少的 log 目录创建 Log。
- `ioThreads`: 为完成 Log 加载的相关操作，每个 log 目录下分配指定的线程执行加载。
- `scheduler`: `KafkaScheduler` 对象，用于执行周期任务的线程池。与 `LogSegment` 小节介绍的执行 `flush()` 操作的 `scheduler` 是同一个对象。
- `logs`: `Pool[TopicAndPartition, Log]` 类型，用于管理 `TopicAndPartition` 与 Log 之间

的对应关系。使用的是 Kafka 自定义的 Pool 类型对象，底层使用 JDK 提供的线程安全的 HashMap——ConcurrentHashMap 实现。

- **dirLocks**: FileLock 集合。这些 FileLock 用来在文件系统层面为每个 log 目录加文件锁。在 LogManager 对象初始化时，就会将所有 log 目录加锁。
- **recoveryPointCheckpoints**: Map[File, OffsetCheckpoint] 类型，用于管理每个 log 目录与其下的 RecoveryPointCheckpoint 文件之间的映射关系。在 LogManager 对象初始化时，会在每个 log 目录下创建一个对应的 RecoveryPointCheckpoint 文件。此 Map 的 value 是 OffsetCheckpoint 类型的对象，其中封装了对应 log 目录下的 RecoveryPointCheckpoint 文件，并提供对 RecoveryPointCheckpoint 文件的读写操作。RecoveryPointCheckpoint 文件中则记录了该 log 目录下的所有 Log 的 recoveryPoint。
- **logCreationOrDeletionLock**: 创建或删除 Log 时需要加锁进行同步。

LogManager 中的定时任务

在 LogManager.startup() 方法中，将三个周期性任务提交到 scheduler 中定时执行，并启动 LogCleaner 线程。LogManager.startup() 方法的实现如下：

```
def startup() {
  if (scheduler != null) { // 下面启动 3 个定时任务
    // 启动 log retention 任务
    scheduler.schedule("kafka-log-retention", cleanupLogs, delay =
InitialTaskDelayMs,
      period = retentionCheckMs, TimeUnit.MILLISECONDS)

    // 启动 log flusher 任务
    scheduler.schedule("kafka-log-flusher", flushDirtyLogs,
      delay = InitialTaskDelayMs, period = flushCheckMs, TimeUnit.
MILLISECONDS)

    // 启动 recovery point checkpoint 任务
    scheduler.schedule("kafka-recovery-point-checkpoint",
      checkpointRecoveryPointOffsets, delay = InitialTaskDelayMs, period =
flushCheckpointMs, TimeUnit.MILLISECONDS)
  }
}
```

```

if (cleanerConfig.enableCleaner)
    cleaner.startup() // 根据 log.cleaner.enable 配置来决定是否启动 LogCleaner
}

```

三个周期性任务的功能如表 4-1 所示。

表 4-1

任务名称	核心方法	任务描述
log-retention 任务	cleanupLogs()	按照两个条件进行 LogSegment 的清理工作：一是 LogSegment 的存活时长，二是整个 Log 的大小。log-retention 任务不仅会将过期的 LogSegment 删除，还会根据 Log 的大小决定是否删除最旧的 LogSegment，以控制整个 Log 的大小。这两个配置在第 1 章的配置文件有所描述，请读者参考
log-flusher 任务	flushDirtyLogs()	根据配置的时长定时对 Log 进行 flush 操作，保证数据的持久性
recovery-point-checkpoint 任务	checkpointRecoveryPointOffsets()	定时将每个 Log 的 recoveryPoint 写入 RecoveryPointCheckpoint 文件中

log-retention 任务通过周期性地调用 `LogManager.cleanupLogs()` 方法完成对符合条件的 LogSegment 的删除。cleanupLogs() 方法的代码如下：

```

def cleanupLogs() {
    var total = 0
    val startMs = time.milliseconds
    // 注意循环的过滤条件，如果 Log 的 cleanup.policy 配置项不为 delete，则不会进行删除
    for (log <- allLogs; if !log.config.compact) {
        // 将任务委托给了 cleanupExpiredSegments() 和 cleanupSegmentsToMaintainSize() 方法
        total += cleanupExpiredSegments(log) + cleanupSegmentsToMaintainSize(log)
    }
}

```

`LogManager.cleanupExpiredSegments()` 方法会根据 LogSegment 的存活时长判断是否要删除 LogSegment。


```

private def cleanupExpiredSegments(log: Log): Int = {
  // 边界检查 (略)
  val startMs = time.milliseconds
  // LogManager 直接管理的是 Log, 但不能直接管理 LogSegment, 所以将删除 LogSegment
  // 的任务交给 Log 处理。删除条件是 LogSegment 的日志文件在最近的一段时间( retentionMs )
  // 内没有被修改
  log.deleteOldSegments(startMs - _.lastModified > log.config.retentionMs)
}

// 下面是 Log.deleteOldSegments() 方法的代码, 对于非 activeSegment 且符合
// predicate 条件的 LogSegment 都会删除
def deleteOldSegments(predicate: LogSegment => Boolean): Int = {
  lock synchronized { // 加锁同步
    val lastEntry = segments.lastEntry // 获取 activeSegment
    val deletable = if (lastEntry == null)
      Seq.empty
    else
      // 通过 logSegments 方法得到 segments 跳表中 value 集合的迭代器, 之后循环检测
      // LogSegment 是否符合删除条件, 并将符合条件的 LogSegment 形成集合返回
      logSegments.takeWhile(s => predicate(s) &&
        (s.baseOffset != lastEntry.getValue.baseOffset || s.size > 0))
    val numToDelete = deletable.size
    if (numToDelete > 0) {
      if (segments.size == numToDelete) // 全部 LogSegment 都符合删除条件
        roll() // 至少保留一个 LogSegment, 删除前, 先创建一个新的 activeSegment
      deletable.foreach(deleteSegment(_)) // 删除 LogSegment
    }
    numToDelete
  }
}

```

最后的 `Log.deleteSegment()` 方法完成了删除 `LogSegment` 的功能, 其主要操作是清除 `segments` 跳表中的 `LogSegment` 对象, 然后将日志文件和索引文件的后缀名改成 “.deleted”, 并创建一个删除这两个文件的任务, 提交到 `scheduler` 线程池中异步执行。`Log.deleteSegment()` 方法的代码如下:

```
private def deleteSegment(segment: LogSegment) {
  lock synchronized {
    segments.remove(segment.baseOffset) // 从 segments 集合中删除 LogSegment 对象
    asyncDeleteSegment(segment) // 异步删除 LogSegment 对应的日志文件和索引文件
  }
}

private def asyncDeleteSegment(segment: LogSegment) {
  // 将日志文件和索引文件的后缀改成 ". deleted"
  segment.changeFileSuffixes("", Log.DeletedFileSuffix)
  def deleteSeg() {
    segment.delete() // 删除日志文件和索引文件
  }
  // deleteSeg() 方法作为定时任务放入 scheduler 线程池里等待执行
  scheduler.schedule("delete-file", deleteSeg, delay = config.
fileDeleteDelayMs)
}
```

介绍完 `LogManager.cleanupExpiredSegments()` 方法之后，再回来看 `LogManager.cleanupSegmentsToMaintainSize()` 方法，它会根据 `retention.bytes` 配置项的值与当前 Log 的大小判断是否删除 `LogSegment`。

```
private def cleanupSegmentsToMaintainSize(log: Log): Int = {
  if (log.config.retentionSize < 0 || log.size < log.config.retentionSize)
    return 0
  var diff = log.size - log.config.retentionSize // 计算需要删除的字节数
  def shouldDelete(segment: LogSegment) = { // 判断该 LogSegment 是否应该被删除
    if (diff - segment.size >= 0) {
      diff -= segment.size
      true
    } else {
      false
    }
  }
  // 调用 Log.deleteOldSegments() 方法，不过删除条件变成了 shouldDelete() 函数
  log.deleteOldSegments(shouldDelete)
}
```

小日志文件的大小，避免磁盘紧张的情况。在很实践场景中，消息的 key 与 value 的

log-flusher 任务会周期性地执行 flush 操作，其执行 flush() 方法的条件只有一个：Log 未刷新时长是否大于此 Log 的 flush.ms 配置项指定的时长。

```
private def flushDirtyLogs() = {
  for ((topicAndPartition, log) <- logs) { // 遍历 logs 集合
    try {
      val timeSinceLastFlush = time.milliseconds - log.lastFlushTime
      if (timeSinceLastFlush >= log.config.flushMs) // 检测是否到时间执行 flush 操作
        log.flush // 调用 Log.flush() 方法，完成刷新操作
    } catch {
      // 异常处理（略）
    }
  }
}
```

在每个 log 目录下都有唯一的一个 RecoveryPointCheckpoint 文件，其中记录此 log 目录下的每个 Log 的 recoveryPoint 值。RecoveryPointCheckpoint 文件会在 Broker 启动时帮助 Broker 进行 Log 的恢复工作，具体恢复操作的流程后面会详细介绍。

recovery-point-checkpoint 任务会周期性地调用 LogManager.checkpointRecoveryPointOffsets() 方法完成 RecoveryPointCheckpoint 文件的更新。checkpointRecoveryPointOffsets() 方法的代码如下：

```
def checkpointRecoveryPointOffsets() {
  // 对每个 log 目录都调用 checkpointLogsInDir() 方法
  this.logDirs.foreach(checkpointLogsInDir)
}

// 下面是 LogManager.checkpointLogsInDir()
private def checkpointLogsInDir(dir: File): Unit = {
  // 获取指定 log 目录下的 TopicAndPartition 信息，以及其对应的 Log 对象
  val recoveryPoints = this.logsByDir.get(dir.toString)

  if (recoveryPoints.isDefined) {
    // 更新指定 log 目录下的 RecoveryPointCheckpoint 文件
    this.recoveryPointCheckpoints(dir).write(
      recoveryPoints.get.mapValues(_.recoveryPoint))
  }
}
```


RecoveryPointCheckpoint 文件的更新操作是在 OffsetCheckpoint 中实现的，其更新方式是：先将 log 目录下的所有 Log 的 recoveryPoint 写到 tmp 文件中，然后用 tmp 文件替换原来的 RecoveryPointCheckpoint 文件。OffsetCheckpoint.write() 方法的是如下：

```
def write(offsets: Map[TopicAndPartition, Long]) {
  lock synchronized { // 加锁
    val fileOutputStream = new FileOutputStream(tempPath.toFile)
    val writer = new BufferedWriter(new OutputStreamWriter(fileOutputStream))

    try {
      writer.write(CurrentVersion.toString) // 写入当前版本号
      writer.newLine()
      writer.write(offsets.size.toString) // 写入记录条数
      writer.newLine()
      // 循环写入 topic 名称、分区编号以及其对应 Log 的 recoveryPoint
      offsets.foreach { case (topicPart, offset) =>
        writer.write(s"${topicPart.topic} ${topicPart.partition} $offset")
        writer.newLine()
      }
      writer.flush()
      fileOutputStream.getFD().sync() // 将写入数据刷新到磁盘
    } catch {
      // 异常处理（略）
    } finally {
      writer.close()
    }
    // 使用 tmp 临时文件替换原来的 RecoveryPointCheckpoint 文件
    Utils.atomicMoveWithFallback(tempPath, path)
  }
}
```

日志压缩

通过上面介绍的 log-retention 任务，Kafka 服务端可以避免出现大量日志占满磁盘的情况。log-retention 任务中配置的阈值非常灵活，可以对整个 Broker 设置全局配置值，也可以对某些特定的 Topic 配置特定值覆盖全局配置。

Kafka 还提供了“日志压缩”（Log Compaction）功能，通过此功能也可以有效地减小日志文件的大小，缓解磁盘紧张的情况。在很多实践场景中，消息的 key 与 value 的值

之间的对应关系是不断变化的，就像数据库中的数据记录会不断被修改一样。如果消费者只关心 key 对应的最新 value 值，可以开启 Kafka 的日志压缩功能，服务端会在后台启动 Cleaner 线程池，定期将相同 key 的消息进行合并，只保留最新的 value 值。日志压缩的原理如图 4-27 所示，这里以 key 值为 key3 的消息为例进行说明，offset 为 3、6、10 的三条消息按时间顺序依次被迫加到 Log 中，在进行日志压缩时，只会保留 key 值为 key3 的最新消息（offset=10）。

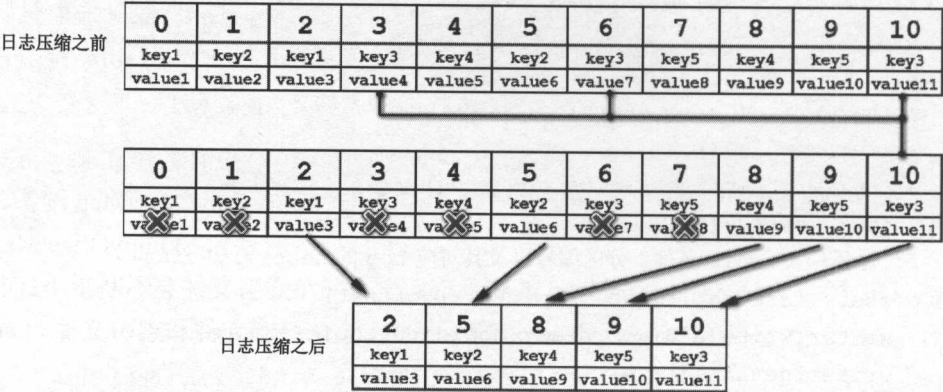


图 4-27

我们已经知道，Log 在写入消息时其实就是将消息追加到 activeSegment 的日志文件末尾。为了避免 activeSegment 成为热点，activeSegment 不会参与日志压缩操作，而是只压缩其余的只读的 LogSegment。在日志压缩过程中启动多条 Cleaner 线程，我们可以通过调整 Cleaner 线程池中的线程数量，优化并发压缩的性能，减少对整体服务端性能的影响。一般情况下，Log 的数据量很大，为了避免 Cleaner 线程与其他业务线程长时间竞争 CPU，并不会将除 activeSegment 之外的所有 LogSegment 在一次压缩操作中全部处理掉，而是将这些 LogSegment 分批进行压缩。

每个 Log 都可以通过 cleaner checkpoint 切分成 clean 和 dirty 两部分，clean 部分表示的是之前已经被压缩过的部分，而 dirty 部分则表示未压缩的部分，如图 4-28 所示。现在假设 Log 中所有的消息都是非压缩消息，所有消息的 offset 都是连续的。日志压缩操作完成后，dirty 部分消息的 offset 依然是连续递增的，而 clean 部分消息的 offset 是断断续续的。cleaner checkpoint 与前面介绍的 Log.recoveryPoint 类似，保存在每个 log 目录对应一个的 cleaner-offset-checkpoint 文件中，由 OffsetCheckpoint 完成相应的读写操作。

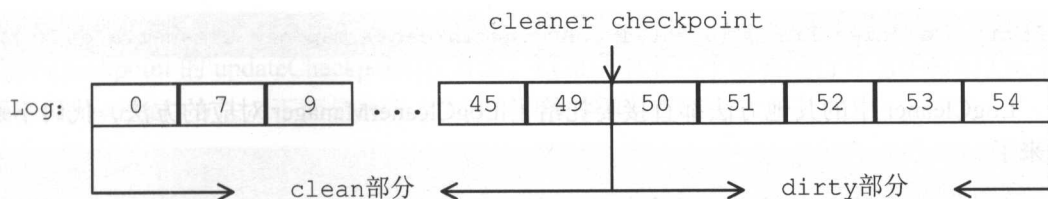


图 4-28

每个 Log 需要进行日志压缩的迫切程度也不同，每个 Cleaner 线程只选取最迫切要被压缩的 Log 进行处理。这里的“迫切程度”是通过 `cleanableRatio`（dirty 部分占整个 Log 的比例）决定的。

Cleaner 线程在选定需要清理的 Log 后，首先为 dirty 部分的消息建立 key 与其 `last_offset`（此 key 出现的最大 offset）的映射关系，该映射通过 `SkimpyOffsetMap` 维护，后面会详细介绍 `SkimpyOffsetMap`。然后重新复制 `LogSegment`，只保留 `SkimpyOffsetMap` 中记录的消息，抛弃掉其他消息。经过日志压缩后，日志文件和索引文件会不断减小，Cleaner 线程还会对相邻的 `LogSegment` 进行合并，避免出现过小的日志文件和索引文件。

最后值得注意的是，在日志压缩时，value 为空的消息会被认为是删除此 key 对应的消息的标志，此标志消息会被保留一段时间，超时后会在下一次日志压缩操作中删除。

介绍完日志压缩的基本概念，来看一下日志压缩相关的实现类之间的依赖关系，如图 4-29 所示。

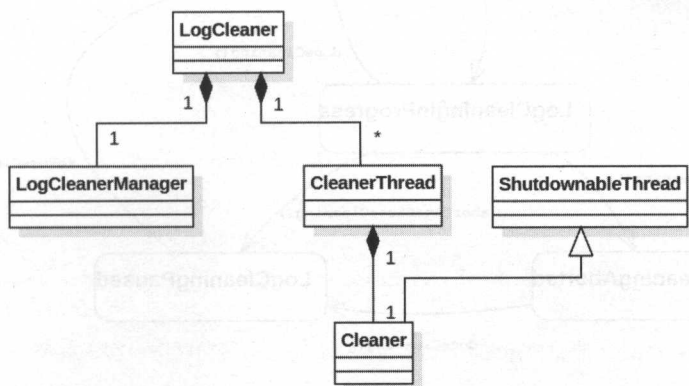


图 4-29

在 `LogCleaner` 中使用 `cleaners` 字段管理 `CleanerThread` 线程，通过 `startup()` 方法和 `shutdown()` 方法完成 `CleanerThread` 线程的启动和停止。

```
private val cleaners = (0 until config.numThreads).map(new CleanerThread(_))
```

LogCleaner 中的其他方法都直接委托给了 LogCleanerManager 对应的方法，代码不贴出来了。

LogCleanerManager 主要负责每个 Log 的压缩状态管理以及 cleaner checkpoint 信息维护和更新。LogCleanerManager 中各个字段的含义如下所述。

- checkpoints: Map[File, OffsetCheckpoint] 类型，用来维护 data 数据目录与 cleaner-offset-checkpoint 文件之间的对应关系，与 LogManager.recoveryPointCheckpoints 集合类似。
- inProgress: HashMap[TopicAndPartition, LogCleaningState] 类型，用于记录正在进行清理的 TopicAndPartition 的压缩状态。

当开始进行日志压缩任务时会先进入 LogCleaningInProgress 状态；压缩任务可以被暂停，此时进入 LogCleaningPaused；压缩任务若被中断，则先进入 LogCleaningAborted 状态，等待 Cleaner 线程将其中的任务中止，然后进入 LogCleaningPaused 状态。处于 LogCleaningPaused 状态的 TopicAndPartition 的日志不会再被压缩，直到有其他线程恢复其压缩状态。压缩状态图如图 4-30 所示。

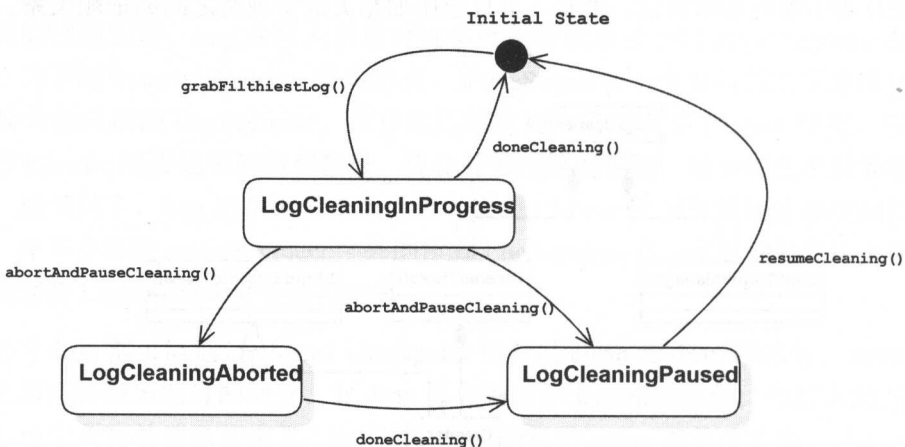


图 4-30

- lock: 用于保护 checkpoints 集合和 inProgress 集合锁。
- pausedCleaningCond: 用于线程阻塞等待压缩状态由 LogCleaningAborted 转换为 LogCleaningPaused。

LogCleanerManager 中除了管理压缩状态的相关方法，还提供了用于修改 cleaner-offset-checkpoint 的 updateCheckpoints() 方法，以及选取下一个需要进行日志压缩的 Log 的 grabFilthiestLog() 方法。下面是这两个方法的相关代码：

```
def updateCheckpoints(dataDir: File, update: Option[(TopicAndPartition,
Long)]) {
  inLock(lock) { // 加锁
    // 获取指定 log 目录对应的 cleaner-offset-checkpoint 文件
    val checkpoint = checkpoints(dataDir)

    // update 会对相同 key 的 value 进行覆盖
    val existing = checkpoint.read().filterKeys(logs.keys) ++ update
    checkpoint.write(existing) // 更新 cleaner-offset-checkpoint 文件
  }
}

// 下面是 grabFilthiestLog() 方法的代码
def grabFilthiestLog(): Option[LogToClean] = {
  inLock(lock) { // 加锁
    val lastClean = allCleanerCheckpoints() // 拿到全部 Log 的 cleaner
    checkpoint
    val dirtyLogs = logs.filter {
      // 过滤掉 cleanup.policy 配置项为 delete 的 Log
      case (topicAndPartition, log) => log.config.compact `
    }.filterNot {
      // 过滤掉 inProgress 包含状态的 Log
      case (topicAndPartition, log) => inProgress.contains(topicAndPartition)
    }.map {
      case (topicAndPartition, log) =>
        // 获取 Log 中第一条消息的 offset
        val logStartOffset = log.logSegments.head.baseOffset

        // 决定最终的压缩开始的位置，firstDirtyOffset 的值可能是 logStartOffset，也可
        // 能是 clean checkpoint
        val firstDirtyOffset = {
          val offset = lastClean.getOrElse(topicAndPartition, logStartOffset)
          if (offset < logStartOffset) {
```



```

        logStartOffset
    } else {
        offset
    }
}

// 为每个 Log 创建一个 LogToClean 对象, 在 LogToClean 对象中维护了每个 Log
// 的 clean 部分字节数、dirty 部分字节数以及 cleanableRatio
LogToClean(topicAndPartition, log, firstDirtyOffset)
}.filter(ltc => ltc.totalBytes > 0) // 过滤掉空 Log

// 获取 dirtyLogs 集合中 cleanableRatio 的最大值
this.dirtiestLogCleanableRatio =
    if (!dirtyLogs.isEmpty) dirtyLogs.max.cleanableRatio else 0

// 过滤掉 cleanableRatio 小于配置的 minCleanableRatio 值的 Log
val cleanableLogs = dirtyLogs.filter(
    ltc => ltc.cleanableRatio > ltc.log.config.minCleanableRatio)
if (cleanableLogs.isEmpty) {
    None
} else {
    val filthiest = cleanableLogs.max // 选择要压缩的 Log

    // 更新 (或添加) 此分区对应的压缩状态
    inProgress.put(filthiest.topicPartition, LogCleaningInProgress)
    Some(filthiest) // 返回要压缩的日志对应的 LogToClean 对象
}
}
}

```

CleanerThread 是执行日志压缩的操作线程, 真正的日志压缩逻辑在其中封装的 Cleaner 对象中。CleanerThread 中的 cleaner 字段定义如下:

```

val cleaner = new Cleaner(id = threadId,
    offsetMap = new SkimpyOffsetMap(memory = math.min(config.dedupeBufferSize /
        config.numThreads, Int.MaxValue).toInt, hashAlgorithm = config.
hashAlgorithm),
    ioBufferSize = config.ioBufferSize / config.numThreads / 2,
    maxIoBufferSize = config.maxMessageSize,
    dupBufferLoadFactor = config.dedupeBufferLoadFactor,
    throttler = throttler, time = time, checkDone = checkDone)

```

这里有必要介绍一下 Cleaner 构造函数的几个重要参数：第二个参数是一个 SkimpyOffsetMap 类型的 Map，其功能在前面提到过，是为 dirty 部分的消息建立 key 与 last_offset 的映射关系。为了减小创建对象的开销，SkimpyOffsetMap 使用加密算法（例如 MD2、MD5、SHA-1 等）计算 key 的 hash 值，并以此 hash 值表示 key，在 Map 中不直接保存 key 的值。SkimpyOffsetMap 底层使用 ByteBuffer 实现，冲突解决方式使用的是线性探查法。SkimpyOffsetMap 只支持 put() 方法和 get() 方法，并不支持任何删除方法，第三个参数 ioBufferSize 指定了读写 LogSegment 的 ByteBuffer 大小，第四个参数 maxIoBufferSize 指定的消息的最大长度，第五个参数 dupBufferLoadFactor 指定了 SkimpyOffsetMap 的最大占用比例，最后一个参数 checkDone 用来检测 Log 的压缩状态。

CleanerThread 继承了 ShutdownableThread，在 ShutdownableThread 中使用 AtomicBoolean 标识其运行状态，并且提供了阻塞等待线程结束的 awaitShutdown() 方法，此方法底层是通过 CountdownLatch 实现的。ShutdownableThread 实现 run() 方法并重新提供了 doWork() 抽象方法供子类实现。CleanerThread.doWork() 方法的实现如下：

```

override def doWork() {
    cleanOrSleep()
}

private def cleanOrSleep() {
    // 通过 grabFilthiestLog() 方法获取需要进行日志压缩的 Log
    cleanerManager.grabFilthiestLog() match {
        case None =>
            // 没有需要压缩的 Log，CleanerThread 线程会睡眠一段时间后，重新执行 doWork() 方法
            backOffWaitLatch.await(config.backOffMs, TimeUnit.MILLISECONDS)
        case Some(cleanable) =>
            var endOffset = cleanable.firstDirtyOffset
    }
}

```

```

try {
    endOffset = cleaner.clean(cleanable) // 调用 Cleaner 对象进行日志压缩
} catch {
    // 异常处理 (略)
} finally {
    // 对 Log 的压缩状态进行转化, 同时更新 cleaner-offset-checkpoint 文件
    cleanerManager.doneCleaning(cleanable.topicPartition,
        cleanable.log.dir.getParentFile(), endOffset)
}
}
}

```

Cleaner.clean() 是日志压缩的入口函数, 其核心步骤如下:

(1) 首先, 确定日志压缩的最大 offset 上限 upperBoundOffset。

(2) 从 firstDirtyOffset 开始遍历 LogSegment, 并填充 OffsetMap。在 OffsetMap 中记录每个 key 应该保留的消息的 offset。当 OffsetMap 被填充满时, 就可以确定日志压缩的实际上限 endOffset。

(3) 根据 deleteRetentionMs 配置, 计算可以安全删除的“删除标识”(即 value 为空的 message) 的 LogSegment。

(4) 将 logStartOffset 到 endOffset 之间的 LogSegment 进行分组, 并按照分组进行日志压缩。

为了便于读者理解, 给出图 4-31, 读者可以结合此图一起完成下面对 Cleaner.clean() 方法中每个步骤实现的分析。

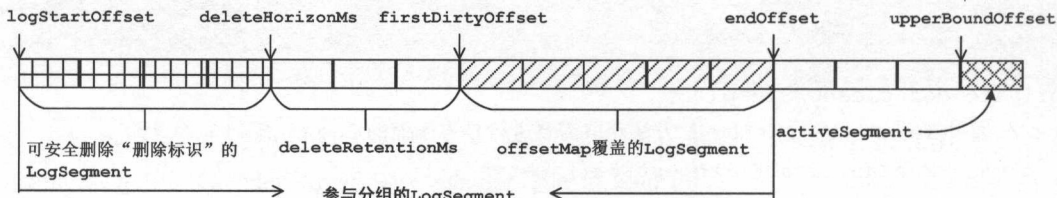


图 4-31

Cleaner.clean() 方法的代码如下:

```
private[log] def clean(cleanable: LogToClean): Long = {
  val log = cleanable.log
  // 步骤 1: 因为 activeSegment 不参与日志压缩, 所以可以确定日志压缩得最大
  // 上限是 activeSegment.baseOffset
  val upperBoundOffset = log.activeSegment.baseOffset

  // 步骤 2: 填充 OffsetMap, 确定日志压缩的真正上限
  val endOffset = buildOffsetMap(log, cleanable.firstDirtyOffset,
    upperBoundOffset, offsetMap) + 1

  // 步骤 3: 计算可以安全删除“删除标识”(即 value 为空的消息)的 LogSegment
  val deleteHorizonMs =
    log.logSegments(0, cleanable.firstDirtyOffset).lastOption match {
      case None      => 0L
      case Some(seg) => seg.lastModified - log.config.deleteRetentionMs
    }

  // 步骤 4: 对要压缩的 Segment 进行分组, 并按照分组进行 clean
  for (group <- groupSegmentsBySize(log.logSegments(0, endOffset),
    log.config.segmentSize, log.config.maxIndexSize))
    cleanSegments(log, group, offsetMap, deleteHorizonMs)
  endOffset
}
```

下面我们对 `Cleaner.clean()` 中的方法进行详细分析。`buildOffsetMap()` 方法主要负责从 `firstDirtyOffset` 开始遍历 `LogSegment`, 并填充 `OffsetMap`。在 `Log` 中写入新消息使用的是追加方式, 那么在填充 `OffsetMap` 过程中, 之后追加到 `Log` 中的消息的 `offset` 会覆盖之前追加到 `Log` 中的消息 `offset`, 因为 `key` 是相同的。当 `OffsetMap` 填满后, 就确定了此次日志压缩的结束位置 `endOffset`。`Cleaner.buildOffsetMap()` 方法的代码如下:

```
private[log] def buildOffsetMap(log: Log, start: Long, end: Long, map:
  OffsetMap): Long = {
  map.clear()
  // 查找从 start ~ end 所有的 LogSegment
  val dirty = log.logSegments(start, end).toBuffer
  var offset = dirty.head.baseOffset
  var full = false // 表示 OffsetMap 是否被填满了
```



```

// 循环遍历 dirty 集合, 循环条件是 OffsetMap 未填满
for (segment <- dirty if !full) {
    // 检查 LogCleanerManager 记录的该分区的压缩状态
    checkDone(log.topicAndPartition)
    // 处理单个 LogSegment, 将消息的 key 和 offset 添加到 OffsetMap 中
    val newOffset = buildOffsetMapForSegment(log.topicAndPartition,
segment, map)
    if (newOffset > -1L) // offsetMap 未满, 移动 offset
        offset = newOffset
    else {
        // 检测 OffsetMap 是否已经填满了。如果 offset>start, 则表示 OffsetMap 中
        // 记录了最后一个 LogSegment 的一部分消息
        require(offset > start, "...")
        full = true // 标识 OffsetMap 满了
    }
}
offset // 返回 offset, 即本次日志压缩的结尾, 不会出现半个 LogSegment 的情况
}

// 下面是 buildOffsetMapForSegment() 方法的具体实现, 在其中完成了对 OffsetMap 的填充
private def buildOffsetMapForSegment(topicAndPartition: TopicAndPartition,
    segment: LogSegment, map: OffsetMap): Long = {
    var position = 0
    var offset = segment.baseOffset
    val maxDesiredMapSize = (map.slots * this.dupBufferLoadFactor).toInt
    while (position < segment.log.sizeInBytes) { // 遍历 LogSegment
        checkDone(topicAndPartition) // 检查压缩状态
        readBuffer.clear()

        // 从 LogSegment 中读取消息
        val messages = new ByteBufferMessageSet(
            segment.log.readInto(readBuffer, position))
        val startPosition = position

        for (entry <- messages) { // 遍历 ByteBufferMessageSet (深层迭代)
            val message = entry.message
            if (message.hasKey) { // 只处理有 key 的消息

```

```

        if (map.size < maxDesiredMapSize)
            // 将 key 和 offset 放入 OffsetMap 中
            map.put(message.key, entry.offset)
        else {
            return -1L // OffsetMap 满了
        }
    }
    offset = entry.offset
}
position += messages.validBytes // 移动 position, 准备下次读取
// position 没有移动, 表示没有读取到一个完整的 Message, 对 readBuffer 和
// writeBuffer 进行扩容, 之后重新读取
if (position == startPosition)
    growBuffers()
}
restoreBuffers() // 重置 readBuffer 和 writeBuffer 的大小
offset // 返回此 LogSegment 的最后一个消息的 offset
}
}

```

确定此次日志压缩的结束位置 `endOffset` 后, 通过 `groupSegmentsBySize()` 方法对 0 到 `endOffset` 的消息进行分组, 分组的单位是 `LogSegment`, 即每个 `LogSegment` 只属于一组, 一个 `LogSegment` 不会被切分到多个组中。在分组过程中, 除了限制了每个分组中消息的总字节数, 还限制了每个分组对应索引的总字节数。`groupSegmentsBySize()` 方法的逻辑比较简单, 请读者参考源码学习。

`groupSegmentsBySize()` 方法返回的 `LogSegment` 分组集合会传入 `cleanSegments()` 方法进行日志压缩。`cleanSegments()` 方法首先创建“.clean”后缀的日志文件和索引文件, 以及对应的 `LogSegment` 对象。之后对一个分组的 `LogSegment` 进行日志压缩, 将压缩后的消息写入“.clean”文件并生成索引项。然后, 将文件的后缀名由“.clean”修改为“.swap”, 并将对应的 `LogSegment` 对象加入到 `segments` 跳表中管理。最后, 将分组中的 `LogSegment` 从 `segments` 删除掉, 将文件的“.swap”后缀名删除掉。`cleanSegments()` 方法的代码如下:

```

private[log] def cleanSegments(log: Log, segments: Seq[LogSegment], map:
OffsetMap,
    deleteHorizonMs: Long) {
// 创建 “.clean” 后缀的日志文件和索引文件，文件名是分组中第一个 LogSegment 的 baseOffset
    val logFile = new File(segments.head.log.file.getPath + Log.
CleanedFileSuffix)
    logFile.delete()
    val indexFile = new File(segments.head.index.file.getPath + Log.
CleanedFileSuffix)
    indexFile.delete()

// 创建对应的 FileMessageSet 对象、OffsetIndex 对象以及 LogSegment 对象
    val messages = new FileMessageSet(logFile, fileAlreadyExists = false,
        initFileSize = log.initFileSize(), preallocate = log.config.
preallocate)
    val index = new OffsetIndex(indexFile, segments.head.baseOffset,
        segments.head.index.maxIndexSize)
    val cleaned = new LogSegment(messages, index, segments.head.baseOffset,
        segments.head.indexIntervalBytes, log.config.randomSegmentJitter,
time)

    try {
        for (old <- segments) {
            // 判定此 LogSegment 中的“删除标记”是否可以安全删除
            val retainDeletes = old.lastModified > deleteHorizonMs
            // 进行日志压缩操作
            cleanInto(log.topicAndPartition, old, cleaned, map, retainDeletes,
                log.config.messageFormatVersion.messageFormatVersion)
        }

        index.trimToValidSize() // 截断多余的索引项
        cleaned.flush() // 执行 flush 操作，将数据刷新到磁盘上
        // 更新最后修改时间
        val modified = segments.last.lastModified
        cleaned.lastModified = modified
    }
}

```



```

// 在Log.replaceSegments()方法中首先会将文件的后缀名由“.clean”修改为“.swap”，
// 并将 cleaned 对象加入到 segments 跳表中管理
// 之后，将分组中的 LogSegment 从 segments 中删除
// 最后将文件的“.swap”后缀名删除
log.replaceSegments(cleaned, segments)
} catch {
// 异常处理（略）
}
}
}

```

最后，我们来分析 `cleanInto()` 方法是如何对一个 `LogSegment` 分组进行日志压缩的。

```

private[log] def cleanInto(topicAndPartition: TopicAndPartition,
                           source: LogSegment, dest: LogSegment, map: OffsetMap,
                           retainDeletes: Boolean,
                           messageFormatVersion: Byte) {
  var position = 0
  while (position < source.log.sizeInBytes) { // 遍历待压缩的 LogSegment
    checkDone(topicAndPartition) // 检测压缩状态
    readBuffer.clear()
    writeBuffer.clear()
    // 读取消息
    val messages = new ByteBufferMessageSet(source.log.readInto(readBuffer,
position))
    var messagesRead = 0
    for (entry <- messages.shallowIterator) { // 遍历每个消息，检测它是否应该保留
      val size = MessageSet.entrySize(entry.message)
      if (entry.message.compressionCodec == NoCompressionCodec) {
        /**
         * 在 shouldRetainMessage() 方法中判断是否保留这个消息，有三个条件：
         * 1). 此消息是否含有 key
         * 2). OffsetMap 是否有相同 key 且 offset 更大的消息
         * 3). 此消息是“删除标记”且此 LogSegment 中的“删除标记”可以安全删除
         */
        if (shouldRetainMessage(source, map, retainDeletes, entry)) {
          // 将需要保留的消息写入 writeBuffer

```



```

        ByteBufferMessageSet.writeMessage(writeBuffer, entry.message,
entry.offset)
    }
    messagesRead += 1
} else {
// 对于压缩消息, 使用深层迭代器迭代内部消息, 逻辑与非压缩消息类似 (略)
// 注意, 如果整个外层消息都要保留, 则不需要重新压缩, 直接将整个外层消息写入 writeBuffer
// 如果一个外层消息中, 只有部分内层消息需要保留, 需要对保留的内层重新压缩后写入 writeBuffer
}
}

position += messages.validBytes
// 如果有需要保留的消息, 则将其追加到压缩后的 LogSegment 中
if (writeBuffer.position > 0) {
    writeBuffer.flip()
    val retained = new ByteBufferMessageSet(writeBuffer)
    dest.append(retained.head.offset, retained)
}

// 未读取到一个完整的消息, 表示 readBuffer 过小, 需要扩容
if (readBuffer.limit > 0 && messagesRead == 0)
    growBuffers()
}
restoreBuffers() // 重置 readBuffer 和 writeBuffer
}

```

到这里, Kafka 的日志压缩功能以及其相关实现就介绍完了。下一节将回到 LogManager 中继续分析其初始化过程。

LogManager 初始化

在 LogManager 的初始化过程中, 除了初始化上述三个定时任务日志压缩的组件, 还会完成相关的恢复操作和 Log 加载。首先调用 LogManager.createAndValidateLogDirs() 方法, 保证每个 log 目录都存在并且可读, 代码比较简单, 就不贴出来了。之后会调用 LogManager.loadLogs() 方法加载 log 目录下的所有 Log。这是 LogManager 初始化的重要过程, 其步骤大致如下:

- (1) 为每个 log 目录分配一个有 ioThreads 条线程的线程池, 用来执行恢复操作。

(2) 检测 Broker 上次关闭是否正常，并设置 Broker 的状态。在 Broker 正常关闭时，会创建一个“.kafka_cleanshutdown”的文件，这里就是通过此文件进行判断的。

(3) 载入每个 Log 的 recoveryPoint。

(4) 为每个 Log 创建一个恢复任务，交给线程池处理。

(5) 主线程阻塞等待所有的恢复任务完成。

(6) 关闭所有在步骤 1 中创建的线程池。

LogManager.loadLogs() 方法的代码如下：

```
private def loadLogs(): Unit = {
  // 保存所有 log 目录对应的线程池
  val threadPools = mutable.ArrayBuffer.empty[ExecutorService]
  val jobs = mutable.Map.empty[File, Seq[Future[_]]]

  for (dir <- this.logDirs) {
    // 步骤 1: 遍历所有 log 目录，为每个目录都创建指定的线程数的线程池
    val pool = Executors.newFixedThreadPool(ioThreads)
    threadPools.append(pool)

    // 步骤 2: 检测 Broker 上次是否正常关闭
    val cleanShutdownFile = new File(dir, Log.CleanShutdownFile)
    if (cleanShutdownFile.exists) {
      // Debug 日志输出 (略)
    } else {
      // 修改 brokerState
      brokerState.newState(RecoveringFromUncleanShutdown)
    }

    // 步骤 3: 读取每个 log 目录下的 RecoveryPointCheckpoint 文件，并生成
    // TopicAndPartition 与 recoveryPoint 的对应关系
    var recoveryPoints = Map[TopicAndPartition, Long]()
    try {
      // 载入 recoveryPoints
      recoveryPoints = this.recoveryPointCheckpoints(dir).read
    }
  }
}
```

```

    } catch {
        // 异常处理 (略)
    }

    val jobsForDir = for { // 遍历所有的 log 目录的子文件, 将文件过滤掉, 只保留目录
        dirContent <- Option(dir.listFiles()).toList
        logDir <- dirContent if logDir.isDirectory
    } yield { // 步骤 4: 为每个 Log 文件夹创建一个 Runnable 任务
        CoreUtils.runnable {
            // 从目录名可以解析出 topic 名称和分区编号
            val topicPartition = Log.parseTopicPartitionName(logDir)
            // 获取 Log 对应的配置
            val config = topicConfigs.getOrElse(topicPartition.topic,
defaultConfig)
            // 获取 Log 对应的 recoveryPoint
            val logRecoveryPoint = recoveryPoints.getOrElse(topicPartition, 0L)
            // 创建 Log 对象
            val current = new Log(logDir, config, logRecoveryPoint, scheduler, time)
            // 将 Log 对象保存到 logs 集合中, 所有分区的 Log 成功加载完成
            val previous = this.logs.put(topicPartition, current)
            if (previous != null) { // 异常处理 (略) }
        }
    }

    // 将 jobsForDir 中的所有任务放到线程池中执行, 并将 Future 形成 Seq, 保存到 jobs 中
    jobs(cleanShutdownFile) = jobsForDir.map(pool.submit).toSeq
}

try {
    // 步骤 5: 等待 jobs 中的 Runnable 完成
    for ((cleanShutdownFile, dirJobs) <- jobs) {
        dirJobs.foreach(_.get)
        cleanShutdownFile.delete() // 删除 cleanShutdownFile 文件
    }
} catch {
    // 异常处理 (略)
} finally {
    threadPools.foreach(_.shutdown()) // 步骤 6: 关闭全部的线程池
}
}

```


从 LogManager.loadLogs() 方法的代码来看，只是创建了 Log 对象，并存入 LogManager.logs 集合进行管理。但是 Log 的初始化过程并不仅仅是创建一个对象而已，它会调用 Log.loadSegments() 方法，其步骤如下：

(1) 删除 “.delete” 和 “.cleaned” 文件。因为存在 “.cleaned” 后缀的文件表示是在日志压缩过程中宕机的，“.cleaned” 文件中数据的状态不明确，无法进行恢复。而 “.swap” 后缀的文件表示日志压缩已经完成了，但在 swap 过程中宕机，“.swap” 文件中保存了日志压缩后的完整消息，可进行恢复。“.delete” 后缀的文件则是本来就要删除的日志文件或索引文件。

```
for (file <- dir.listFiles if file.isFile) {
  if (!file.canRead) throw new IOException("Could not read file " + file)
  val filename = file.getName
  if (filename.endsWith(DeletedFileSuffix)
      || filename.endsWith(CleanedFileSuffix)) {
    file.delete() // 删除 “.delete” 后缀的文件和 “.cleaned” 后缀的文件
  } else if (filename.endsWith(SwapFileSuffix)) { // 处理 “.swap” 后缀的文件
    // 将 “.swap” 后缀去掉
    val baseName = new File(CoreUtils.replaceSuffix(file.getPath,
        SwapFileSuffix, ""))
    if (baseName.getPath.endsWith(IndexFileSuffix)) {
      file.delete() // 去掉 “.swap” 后缀后发现是索引文件，直接删除
    } else if (baseName.getPath.endsWith(LogFileSuffix)) {
      // 去掉 “.swap” 后缀后发现是日志文件，则需要恢复
      val index = new File(CoreUtils.replaceSuffix(baseName.getPath,
          LogFileSuffix, IndexFileSuffix))
      index.delete() // delete the index
      swapFiles += file // 添加到 swapFiles 集合中，待后面恢复
    }
  }
}
```

(2) 加载全部的日志文件和索引文件。如果索引文件没有配对的日志文件，则删除索引文件；如果日志文件没有对应的索引文件，则重建索引文件。此过程代码如下：


```

for (file <- dir.listFiles if file.isFile) { // 遍历所有文件
    val filename = file.getName
    if (filename.endsWith(IndexFileSuffix)) { // 处理索引文件
        val logFile = new File(file.getAbsolutePath
            .replace(IndexFileSuffix, LogFileSuffix))
        if (!logFile.exists()) {
            file.delete() // 清理无效（即无配对的日志文件）的索引文件
        }
    } else if (filename.endsWith(LogFileSuffix)) { // 处理日志文件
        // 对于日志文件，LogSegment 对象完成 load
        val start = filename.substring(0,
            filename.length - LogFileSuffix.length).toLong
        val indexFile = Log.indexFilename(dir, start)
        val segment = new LogSegment(...) // 创建 LogSegment
        if (indexFile.exists()) { // 检测索引文件是否存在
            try {
                segment.index.sanityCheck() // 检查索引文件的完整性
            } catch {
                // 异常处理（略）
                segment.recover(config.maxMessageSize)
            }
        } else {
            // 如果没有对应索引文件，则需要重建索引文件，LogSegment.recover() 方法已经在前面分析过
            segment.recover(config.maxMessageSize)
        }
        // 将 LogSegment 对象放到 segments 跳表中管理
        segments.put(start, segment)
    }
}
}

```

(3) 处理步骤 1 中记录的“.swap”文件，原理与日志压缩最后的步骤类似。

```

for (swapFile <- swapFiles) {
    val logFile = new File(CoreUtils.replaceSuffix(swapFile.getPath,
        SwapFileSuffix, ""))
    val fileName = logFile.getName
    // 根据日志文件的名称得到 baseOffset

```

```

val startOffset = fileName.substring(0,
    fileName.length - LogFileSuffix.length).toLong
val indexFile = new File(CoreUtils.replaceSuffix(logFile.getPath,
    LogFileSuffix, IndexFileSuffix) + SwapFileSuffix)
val index = new OffsetIndex(indexFile, baseOffset = startOffset,
    maxIndexSize = config.maxIndexSize)

// 创建 LogSegment
val swapSegment = new LogSegment(new FileMessageSet(file = swapFile),
    index = index,
    baseOffset = startOffset,
    indexIntervalBytes = config.indexInterval,
    rollJitterMs = config.randomSegmentJitter,
    time = time)
// 重建索引文件并验证日志文件
swapSegment.recover(config.maxMessageSize)
// 查找 swapSegment 对应的日志压缩前的 LogSegment 集合
val oldSegments = logSegments(swapSegment.baseOffset, swapSegment.
nextOffset)
// 在 Log.replaceSegments() 方法中将 swapSegment 对象加入到 segments 跳表中管理
// 将 oldSegments 集合中的全部 LogSegment 对象从 segments 删除掉并删除对应日志文件
// 和索引文件，最后将文件的“.swap”后缀名删除掉
replaceSegments(swapSegment, oldSegments.toSeq, isRecoveredSwapFile =
true)
}

```

(4) 对于空的 Log，需要创建 activeSegment，保证 Log 中至少有一个 LogSegment。而对于非空的 Log，则需要恢复操作。

```

// 上面的几步处理中没有得到任何 LogSegment，表示此 Log 为空，需要创建 activeSegment
if (logSegments.size == 0) {
    segments.put(OL, new LogSegment(...))
} else {
    recoverLog() // 进行 Log 恢复操作
    activeSegment.index.resize(config.maxIndexSize)
}

```

recoverLog() 方法主要负责处理 Broker 非正常关闭时导致的消息异常，需要将

recoveryPoint~activeSegment 中的所有消息进行验证，将验证失败的消息截断。

```
private def recoverLog() {
  // 如果 Broker 上次是正常关闭的，则不需要恢复，更新 recoveryPoint
  if (hasCleanShutdownFile) {
    this.recoveryPoint = activeSegment.nextOffset
    return
  }

  // 如果 Broker 上次是非正常关闭的，则需要恢复操作
  // 获取全部未刷新的 LogSegment，即 recoveryPoint 之后的全部 LogSegment
  val unflushed = logSegments(this.recoveryPoint, Long.MaxValue).iterator
  while (unflushed.hasNext) {
    val curr = unflushed.next
    val truncatedBytes =
      try {
        // 使用 LogSegment.recover() 方法重建索引文件并验证日志文件，验证失败的部分则截掉
        curr.recover(config.maxMessageSize)
      } catch {
        // 异常处理（略）
      }
    if (truncatedBytes > 0) { // LogSegment 中有验证失败的消息
      unflushed.foreach(deleteSegment) // 将剩余的 LogSegment 全部删除
    }
  }
}
```

LogManager 中还提供的三个重要的方法，它们分别是 createLog() 方法、deleteLog() 方法和 getLog() 方法。getLog() 方法代码比较简单，不再贴出来了。createLog() 方法在选择创建 Log 的 log 目录时，会选择 Log 最少的 log 目录，实现如下：

```
def createLog(topicAndPartition: TopicAndPartition, config: LogConfig): Log = {
  logCreationOrDeletionLock synchronized { // 加锁
    var log = logs.get(topicAndPartition)
    ... // 边界检查（略）
    val dataDir = nextLogDir() // 选择 Log 最少的 log 目录
    val dir = new File(dataDir,
      topicAndPartition.topic + "-" + topicAndPartition.partition)
    dir.mkdirs() // 创建 Log 对应的文件夹
  }
}
```



```

// 创建 Log 对象, 根据之前的分析, 这里会同时创建 activeSegment
log = new Log(dir, config, recoveryPoint = 0L, scheduler, time)
logs.put(topicAndPartition, log) // 存入 logs 集合中
log
}
}
private def nextLogDir(): File = {
  if (logDirs.size == 1) { // 只有一个 log 目录
    logDirs(0)
  } else { // 指定了多个 log 目录
    // 计算每个 log 目录中的 Log 数量
    val logCounts = allLogs.groupBy(_.dir.getParent).mapValues(_.size)
    val zeros = logDirs.map(dir => (dir.getPath, 0)).toMap
    var dirCounts = (zeros ++ logCounts).toBuffer

    val leastLoaded = dirCounts.sortBy(_._2).head // 选择 Log 最少的 log 目录
    new File(leastLoaded._1)
  }
}
}

```

在 `LogManager.deleteLog()` 方法中, 需要先停止对当前 Log 的日志压缩操作, 再进行删除, 代码如下:

```

def deleteLog(topicAndPartition: TopicAndPartition) {
  var removedLog: Log = null
  logCreationOrDeletionLock synchronized { // 加锁
    removedLog = logs.remove(topicAndPartition) // 从 logs 集合中对应的 Log 对象
  }
  if (removedLog != null) {
    if (cleaner != null) {
      // 停止对此 Log 的日志压缩操作, 这里会阻塞等待压缩状态
      cleaner.abortCleaning(topicAndPartition)
      // 更新 cleaner-offset-checkpoint 文件
      cleaner.updateCheckpoints(removedLog.dir.getParentFile)
    }
    removedLog.delete() // 删除相关的日志文件、索引文件和目录
  }
}
}

```


Kafka 中关于日志存储的相关内容就介绍完了。本节介绍了 Kafka 日志存储相关的基本概念，为读者描述了日志子系统的工作原理。然后介绍了存储消息的日志文件格式以及索引文件格式，介绍了 `FileMessageSet` 和 `OffsetIndex` 对日志文件和索引文件的管理，分析了服务端的 `ByteBufferMessageSet` 和客户端的 `MemoryRecords` 如何灵活地处理压缩消息。之后介绍了 `LogSegment` 如何管理日志和索引，以及 `Log` 如何管理和操纵 `LogSegment` 对象集合。最后，详细分析了 `LogManager` 的功能，例如，`log-flusher` 定时任务、`log-retention` 定时任务、`recovery-point-checkpoint` 定时任务以及日志压缩的功能，还介绍了 `LogManager` 的恢复操作和加载 `Log` 的流程。希望读者通过本章，能够深入了解 Kafka 日志存储方面的设计和实现。

4.4 DelayedOperationPurgatory 组件

`DelayedOperationPurgatory` 是一个相对独立的组件，它的主要功能是管理延迟操作。`DelayedOperationPurgatory` 的底层依赖于 Kafka 提供的时间轮实现。有的读者可能会感到奇怪，我们可以使用 JDK 本身提供的 `java.util.Timer` 或是 `DelayQueue` 轻松实现定时任务的功能，为什么 Kafka 还要专门开发 `DelayedOperationPurgatory` 组件呢？这主要是因为像 Kafka 这种分布式系统的请求量巨大，性能要求也很高，JDK 提供 `java.util.Timer` 和 `DelayedQueue` 底层实现使用的是堆这种数据结构，存取操作的复杂度都是 $O(n\log(n))$ ，无法支持大量的定时任务。在高性能的框架中，为了将定时任务的存取操作以及取消操作的时间复杂度降为 $O(1)$ ，一般会使用其他方式实现定时任务组件，例如，使用时间轮的方式。这种做法还是比较多见的，例如，`ZooKeeper` 中使用“时间桶”的方式处理 `Session` 过期，`Netty` 也提供了 `HashedWheelTimer` 这种时间轮的实现，`Quartz` 框架中也有时间轮的身影。

4.4.1 TimingWheel

Kafka 的时间轮实现是 `TimingWheel`，它是一个存储定时任务的环形队列，底层使用数组实现，数组中的每个元素可以存放一个 `TimerTaskList` 对象。`TimerTaskList` 是环形双向链表，在其中的链表项 `TimerTaskEntry` 中封装了真正的定时任务 `TimerTask`。`TimerTaskList` 使用 `expiration` 字段记录了整个 `TimerTaskList` 的超时时间。`TimerTaskEntry` 中的 `expirationMs` 字段记录了超时时间戳，`timerTask` 字段指向了对应的 `TimerTask` 任务。`TimerTask` 中的 `delayMs` 记录了任务的延迟时间，`timerTaskEntry` 字段记录了对应的 `TimerTaskEntry` 对象。这三个对象是 `TimingWheel` 实现的基础。

`TimingWheel` 提供了层级时间轮的概念，如图 4-32 所示，第一层时间轮的时间跨度比较小，而第二层时间轮的时间跨度比较大。存放在同一 `TimerTaskList` 中的 `TimerTask` 到期

时间可能不同,但是都由一个时间格覆盖。现假设图中第二层时间轮中编号为2的时间格保存的 TimerTaskList 到期时间为 t ,其中保存的任务的过期时间只能是 $[t \sim t+20\text{ms}]$ 这个范围,例如 T1 的过期时间可以为 $t+10\text{ms}$,任务 T2 的过期时间可以为 $t+15\text{ms}$,T3 的过期时间可以为 $t+12\text{ms}$ 。如果任务到期时间不在 $[t \sim t+20\text{ms}]$ 这个时间段,则只能放到其他的时间格对应的 TimerTaskList 中保存。

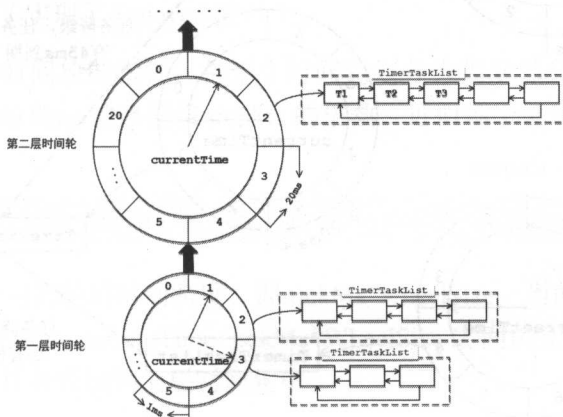


图 4-32

当任务到期时间超出了当前时间轮所表示的时间范围时,会尝试添加到上层时间轮。依然以图 4-32 为例,其中第一层时间轮的每个时间格是 1ms ,整个时间轮的跨度是 20ms ,其表针 `currentTime` 当前表示的时间 ct ,则该时间轮的跨度为 $[ct \sim ct+20\text{ms}]$,只有到期时间在这段范围内的任务才能添加到该时间轮中等待到期。到期时间超出 $[ct \sim ct+20\text{ms}]$ 这个时间范围的任务会尝试添加到其上级时间轮中,通过逐层向上尝试,最终找到合适的时间轮层级。

整个时间轮表示的时间跨度是不变的,随着表针 `currentTime` 的后移,当前时间轮能处理时间段也在不断后移,新来的 `TimerTaskEntry` 会复用原来已经到期的 `TimerTaskList`。如图 4-32 所示,第一层时间轮的时间跨度始终为 20ms ,表针 `currentTime` 表示的时间随着时间的流逝不断后移,指向了第三个时间格,此时表针 `currentTime` 表示的时间为 $ct+3\text{ms}$,整个时间轮表示的时间段是 $[ct+3\text{ms} \sim ct+23\text{ms}]$,但是该时间轮的时间跨度依然是 20ms 。此时该时间轮中编号为 2 的时间格表示的时间范围不再是 $[ct+1\text{ms} \sim ct+2\text{ms}]$,而是 $[ct+22\text{ms} \sim ct+23\text{ms}]$ 。

最后用一个示例详细介绍时间轮降级场景,如图 4-33 所示。

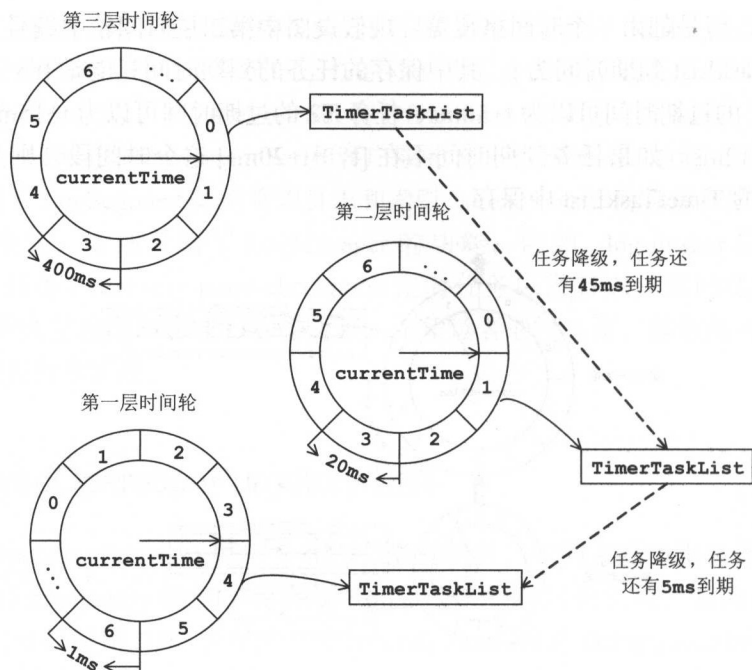


图 4-33

现在有一个任务是在 445ms 后执行, 默认情况下, 各个层级的时间轮的时间格个数为 20, 第一层时间轮每个时间格的跨度为 1ms, 整个时间轮的跨度为 20ms, 跨度不够。第二层时间轮时间格的跨度为 20ms, 整个时间轮的跨度为 400ms, 跨度依然不够。第三层时间轮时间格的跨度为 400ms, 整个时间轮的跨度为 8000ms, 跨度足够, 此任务存放在第三层时间轮的第一个时间格对应的 `TimerTaskList` 中等待执行, 此 `TimerTaskList` 到期时间是 400ms。随着时间的流逝, 当此 `TimerTaskList` 到期时, 距离该任务的到期时间还有 45ms, 不能执行任务。我们将其重新提交到层级时间轮中, 此时第一层时间轮跨度依然不够, 但是第二层时间轮的跨度足够, 该任务会被放到第二层时间轮第三个时间格中等待执行。如此往复几次, 高层时间轮的任务会慢慢移动到低层时间轮上, 最终任务到期执行。

介绍完了时间轮的基本概念之后, 下面开始分析 `TimingWheel` 的具体实现, 其核心字段的含义如下所述。

- `buckets`: `Array.tabulate[TimerTaskList]` 类型, 其每一个项都对应时间轮中的一个时间格, 用于保存 `TimerTaskList` 的数组。在 `TimingWheel` 中, 同一个 `TimerTaskList` 中的不同定时任务的到期时间可能不同, 但是相差时间在一个时间格的范围内。
- `tickMs`: 当前时间轮中一个时间格表示的时间跨度。

- wheelSize: 当前时间轮的格数, 也是 buckets 数组的大小。
- taskCounter: 各层级时间轮中任务的总数。
- startMs: 当前时间轮的创建时间。
- queue: DelayQueue 类型, 整个层级时间轮共用的一个任务队列, 其元素类型是 TimerTaskList (实现了 Delayed 接口)。
- currentTime: 时间轮的指针, 将整个时间轮划分为到期部分和未到期部分。在初始化时, currentTime 被修剪成 tickMs 的倍数, 近似等于创建时间, 但并不是严格的创建时间。

```
private[this] var currentTime = startMs - (startMs % tickMs)
```

- interval: 当前时间轮的时间跨度, 即 tickMs*wheelSize。当前时间轮只能处理时间范围在 currentTime~currentTime+tickMs*WheelSize 之间的定时任务, 超过这个范围, 则需要将任务添加到上层时间轮中。
- overflowWheel: 上层时间轮的引用。

在 TimeWheel 中提供了 add()、advanceClock()、addOverflowWheel() 三个方法, 这三个方法实现了时间轮的基础功能。add() 方法实现了向时间轮中添加定时任务的功能, 它同时也会检测待添加的任务是否已经到期。

```
def add(timerTaskEntry: TimerTaskEntry): Boolean = {
  val expiration = timerTaskEntry.expirationMs
  if (timerTaskEntry.cancelled) { // 任务已经被取消
    false
  } else if (expiration < currentTime + tickMs) { // 任务已经到期
    false
  } // 任务在当前时间轮的跨度范围内
  } else if (expiration < currentTime + interval) {
    // 按照任务的到期时间查找此任务属于的时间格, 并将任务添加到对应的 TimerTaskList 中
    val virtualId = expiration / tickMs
    val bucket = buckets((virtualId % wheelSize.toLong).toInt)
    bucket.add(timerTaskEntry)
    // 整个时间轮表示的时间跨度是不变的, 随着表针 currentTime 的后移, 当前时间轮能处理
    // 时间段也在不断后移, 新来的 TimerTaskEntry 会复用原来已经清理过的
    // TimerTaskList (bucket)。此时需要重置 TimerTaskList 的到期时间, 并将 bucket
    // 重新添加到 DelayQueue 中。后面还会介绍这个 DelayQueue 的作用
```



```

        if (bucket.setExpiration(virtualId * tickMs)) { // 设置 bucket 的到期时间
            queue.offer(bucket)
        }
        true
    } else { // 超出了当前时间轮的时间跨度范围, 则将任务添加到上层时间轮中处理
        if (overflowWheel == null)
            addOverflowWheel() // 创建上层时间轮
        overflowWheel.add(timerTaskEntry)
    }
}

```

`addOverflowWheel()` 方法会创建上层时间轮, 默认情况下, 上层时间轮的 `tickMs` 是当前整个时间轮的时间跨度 `interval`。

```

private[this] def addOverflowWheel(): Unit = {
    synchronized {
        if (overflowWheel == null) {
            // 创建上层时间轮, 注意, 上层时间轮的 tickMs 更大, wheelSize 不变, 则表示的时间
            // 跨度也就越大
            // 随着上层时间轮表针的转动, 任务还是会回到最底层的时间轮上, 等待最终超时
            overflowWheel = new TimingWheel(
                tickMs = interval,
                wheelSize = wheelSize,
                startMs = currentTime,
                taskCounter = taskCounter, // 全局唯一的任务计数器
                queue // 全局唯一的任务队列
            )
        }
    }
}

```

`advanceClock()` 方法会尝试推进当前时间轮的表针 `currentTime`, 同时也会尝试推进上层的时间轮的表针。随着当前时间轮的表针不断被推进, 上层时间轮的表针也早晚会被推进成功。

```
def advanceClock(timeMs: Long): Unit = {
    // 尝试移动表针 currentTime, 推进可能不止一格
    if (timeMs >= currentTime + tickMs) {
        currentTime = timeMs - (timeMs % tickMs)
        // 尝试推进上层时间轮的表针
        if (overflowWheel != null)
            overflowWheel.advanceClock(currentTime)
    }
}
```

4.4.2 SystemTimer

SystemTimer 是 Kafka 中的定时器实现, 它在 TimeWheel 的基础上添加了执行到期任务、阻塞等待最近到期任务的功能。下面来分析其核心字段。

- taskExecutor: JDK 提供的固定线程数的线程池实现, 由此线程池执行到期任务。
- delayQueue: 各个层级的时间轮共用的 DelayQueue 队列, 主要作用是阻塞推进时间轮表针的线程 (ExpiredOperationReaper), 等待最近到期任务到期。
- taskCounter: 各个层级时间轮共用的任务个数计数器。
- timingWheel: 层级时间轮中最底层的时间轮。
- readWriteLock: 用来同步时间轮表针 currentTime 修改的读写锁。

SystemTimer.add() 方法在添加过程中如果发现任务已经到期, 则将任务提交到 taskExecutor 中执行; 如果任务未到期, 则调用 TimeWheel.add() 方法提交到时间轮中等待到期后执行。SystemTimer.add() 方法的实现如下:

```
def add(timerTask: TimerTask): Unit = {
    readLock.lock()
    try {
        // 将 TimerTask 封装成 TimerTaskEntry, 并计算其到期时间
        addTimerTaskEntry(new TimerTaskEntry(timerTask,
            timerTask.delayMs + System.currentTimeMillis()))
    } finally {
        readLock.unlock()
    }
}
```

```
private def addTimerTaskEntry(timerTaskEntry: TimerTaskEntry): Unit = {
  // 向时间轮提交添加任务失败，任务可能已到期或已取消
  if (!timingWheel.add(timerTaskEntry)){
    if (!timerTaskEntry.cancelled) // 将到期任务提交到 taskExecutor 执行
      taskExecutor.submit(timerTaskEntry.timerTask)
  }
}
```

SystemTimer.advanceClock() 方法完成了时间轮表针的推进，同时对到期的 TimerTaskList 中的任务进行处理。如果 TimerTaskList 到期，但是其中的某些任务未到期，会将未到期任务进行降级，添加到低层次的时间轮中继续等待；如果任务到期了，则提交到 taskExecutor 线程池中执行。

```
def advanceClock(timeoutMs: Long): Boolean = {
  var bucket = delayQueue.poll(timeoutMs, TimeUnit.MILLISECONDS) // 阻塞等待
  if (bucket != null) { // 在阻塞期间，有 TimerTaskList 到期
    writeLock.lock()
    try {
      while (bucket != null) {
        timingWheel.advanceClock(bucket.getExpiration()) // 推进时间轮表针
        // 调用 reinsert，尝试将 bucket 中的任务重新添加到时间轮。此过程并不一定是将任
        // 务提交给 taskExecutor 执行，对于未到期的任务只是从原来的时间轮降级到下层的
        // 时间轮继续等待
        bucket.flush(reinsert)
        bucket = delayQueue.poll() // 此 poll() 方法不会阻塞
      }
    } finally {
      writeLock.unlock()
    }
    true
  } else {
    false
  }
}

// TimerTaskEntry 重新提交到时间轮中
private[this] val reinsert =
  (timerTaskEntry: TimerTaskEntry) => addTimerTaskEntry(timerTaskEntry)
```

4.4.3 DelayedOperation

在前面介绍 `ProducerRequest` 和 `FetchRequest` 两种请求时提到，服务端在收到这两种请求时并不是立即返回响应，可能会等待一段时间后才返回。对于 `ProducerRequest` 来说，其中的 `acks` 字段设置为 `-1` 表示 `ProducerRequest` 发送到 Leader 副本之后，需要 ISR 集合中所有副本都同步该请求中的消息（或超时）后，才能返回响应给客户端。ISR 集合中的副本分布在不同 Broker 上，与 Leader 副本进行同步时就涉及网络通信，一般情况下我们认为网络传输是不可靠的而且是一个较慢的过程，通常采用异步的方式处理来避免线程长时间等待。当 `FetchRequest` 发送给 Leader 副本后，会积累一定量的消息后才返回给消费者或 Follower 副本，并不是 Leader 副本的 HW 后移一条消息就立即将其返回给消费者，这是为了实现批量发送消息，提高有效负载。

Kafka 利用前面介绍的 `SystemTimer` 来定期检测请求是否超时，但是这些请求真正的目的并不是为了超时执行，而是为了满足其他条件后执行，例如 `ProducerRequest` 的响应条件 ISR 集合中所有副本都同步了请求中的消息，所以仅使用 `SystemTimer` 就无法满足需求了。Kafka 使用 `DelayedOperation` 抽象类表示延迟操作，它对 `TimeTask` 进行了扩展，除了有定时执行的功能，还提供了检测其他执行条件的功能。我们可以认为 `DelayedOperation` 是一个延迟的、异步的操作。

如图 4-34 所示，`DelayedOperation` 有四个实现类，分别表示四类不同的延迟操作，也对应了四种不同的请求。`DelayedOperation` 是一个抽象类，`completed` 字段是 `AtomicBoolean` 类型，标识了此 `DelayedOperation` 是否完成，初始值为 `false`；`delayMs` 记录了延迟操作的延迟时长。

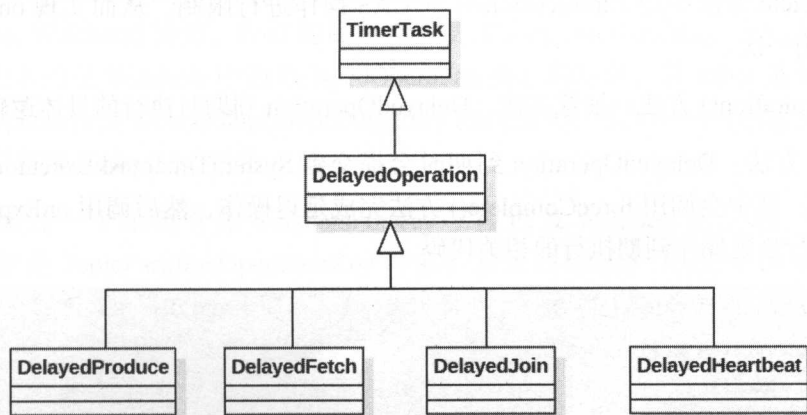


图 4-34

DelayedOperation 中各个方法的含义和功能如下所述。

- `onComplete()` 方法：抽象方法，DelayedOperation 的具体业务逻辑。例如 DelayedProduce 中该方法的实现就是向客户端返回 ProduceResponse 响应。此方法只能在 `forceComplete()` 方法中被调用，且在 DelayedOperation 的整个生命周期中只能被调用一次。
- `forceComplete()` 方法：如果 DelayedOperation 没有完成，则先将任务从时间轮中删除掉，然后调用 `onComplete()` 方法执行其具体的业务逻辑。

```
def forceComplete(): Boolean = {
  // 根据 completed 字段的值判断延迟操作是否已经完成
  if (completed.compareAndSet(false, true)) {
    // 调用的是 TimerTask.cancel() 方法，将其从 TimerTaskList 中删除
    cancel()
    // 延迟操作的真正逻辑，例如，DelayProduce 就是向客户端返回 ProduceResponse 响应
    onComplete()
    true
  } else {
    false
  }
}
```

可能有多个 Handler 线程并发检测 DelayedOperation 的执行条件，这就可能导致多个线程并发调用 `forceComplete()` 方法，但是 `onComplete()` 方法有只能调用一次的限制。因此在 `forceComplete` 方法中用 AtomicBoolean 的 CAS 操作进行限制，从而实现 `onComplete()` 方法只被调用一次。

- `onExpiration()` 方法：抽象方法，DelayedOperation 到期时执行的具体逻辑。
- `run()` 方法：DelayedOperation 到期时会提交到 SystemTimer.taskExecutor 线程池中执行。其中会调用 `forceComplete()` 方法完成延迟操作，然后调用 `onExpiration()` 方法执行延迟操作到期执行的相关代码。

```
override def run(): Unit = {
  if (forceComplete())
    onExpiration()
}
```

- `tryComplete()` 方法：抽象方法，在该方法中子类会根据自身的具体类型，检测执

行条件是否满足，若满足则会调用 `forceComplete()` 完成延迟操作。

- `isCompleted()` 方法：检测任务是否完成。

`DelayedOperation` 可能因为到期而被提交到 `SystemTimer.taskExecutor` 线程池中执行，也可能在其他线程检测其执行条件时发现已经满足执行条件，而将其执行。为了读者更好地理解这两条执行路线，给出图 4-35 供读者参考。

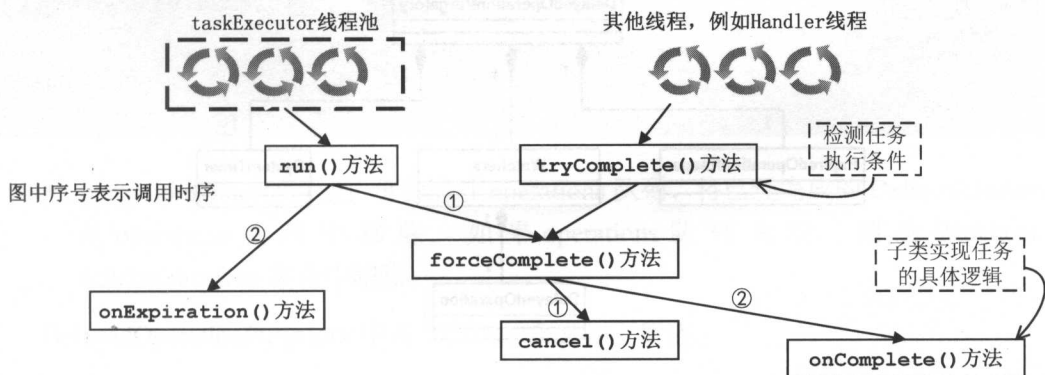


图 4-35

4.4.4 DelayedOperationPurgatory

`DelayedOperationPurgatory` 是一个辅助类，提供了管理 `DelayedOperation` 以及处理到期 `DelayedOperation` 的功能。`DelayedOperationPurgatory` 依赖的组件如图 4-36 所示。

`DelayedOperationPurgatory` 中的 `watchersForKey` 字段用于管理 `DelayedOperation`，它是 `Pool[Any, Watchers]` 类型，`Pool` 的底层实现是 `ConcurrentHashMap`。`watchersForKey` 集合的 `key` 表示的是 `Watchers` 中的 `DelayedOperation` 关心的对象，其 `value` 是 `Watchers` 类型的对象，`Watchers` 是 `DelayedOperationPurgatory` 的内部类，表示一个 `DelayedOperation` 的集合，底层使用 `LinkedList` 实现。

下面通过一个示例介绍 `watchersForKey` 字段以及 `Watchers` 的功能。`DelayProduce` 关心的对象是 `TopicPartitionOperationKey` 对象，表示的是某个 `Topic` 中的某个分区。假设现在有一个 `ProducerRequest` 请求，它要向名为“test”的 `Topic` 中追加消息，分区的编号为 0，此分区当前的 `ISR` 集合中有三个副本。该 `ProducerRequest` 的 `acks` 字段为 -1 表示需要 `ISR` 集合中所有副本都同步了该请求中的消息才能返回 `ProduceResponse`。`Leader` 副本处理此 `ProducerRequest` 时会为其生成一个对应的 `DelayedProduce` 对象，并交给 `DelayedOperationPurgatory` 管理，`DelayedOperationPurgatory` 会将其存放到“test-0”

(TopicPartitionOperationKey 对象)对应的 Watchers 中,同时也会将其提交到 SystemTimer 中。之后,每当 Leader 副本收到 Follower 副本发送的对“test-0”的 FetchRequest 时,都会检测“test-0”对应的 Watchers 中的 DelayedProduce 是否已经满足了执行条件,如果满足执行条件就会执行 DelayedProduce,向客户端返回 ProduceResponse。最终,该 DelayedProduce 会因满足执行条件或时间到期而被执行。

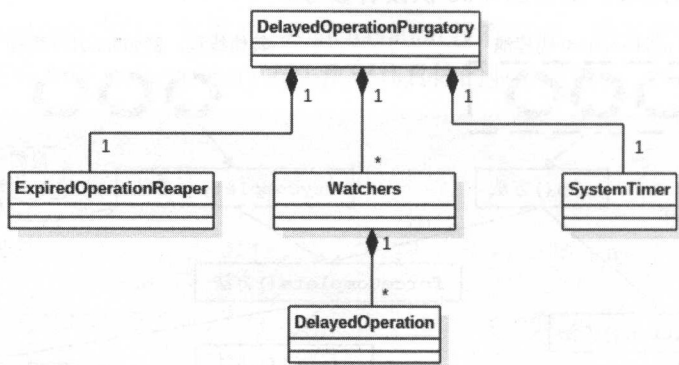


图 4-36

Watchers 的字段只有一个 operations 字段,它用于管理 DelayedOperation 的 LinkedList 队列,下面来分析 Watchers 其核心方法有三个:

- watch() 方法: 将 DelayedOperation 添加到 operations 队列中。
- tryCompleteWatched() 方法: 遍历 operations 队列,对于未完成的 DelayedOperation 执行 tryComplete() 方法尝试完成,将已完成的 DelayedOperation 对象移除。如果 operations 队列为空,则将 Watchers 从 DelayedOperationPurgatory.watchersForKey 中删除。

```

def tryCompleteWatched(): Int = {
    var completed = 0
    operations synchronized {
        val iter = operations.iterator()
        while (iter.hasNext) { // 遍历 operations 队列
            val curr = iter.next()
            // DelayedOperation 已经完成,将其从 operations 队列移除
            if (curr.isCompleted) {
                iter.remove()
            } else if (curr.synchronized curr.tryComplete()) {
                // 调用 DelayedOperation.tryComplete() 方法,尝试完成延迟操作
            }
        }
    }
    completed
}
  
```

```

        completed += 1
        iter.remove() // 完成后将 DelayedOperation 对象从 operations 队列移除
    }
}
}
if (operations.size == 0)
    removeKeyIfEmpty(key, this)

completed
}

```

- `purgeCompleted()` 方法：负责清理 `operations` 队列，将已经完成的 `DelayedOperation` 从 `operations` 队列中移除，如果 `operations` 队列为空，则将 `Watchers` 从 `watchersForKey` 集合中删除。

`DelayedOperationPurgatory` 中各个字段的含义如下所述。

- `timeoutTimer`：前面介绍的 `SystemTimer` 对象。
- `watchersForKey`：管理 `Watchers` 的 `Pool` 对象。
- `removeWatchersLock`：对 `watchersForKey` 进行同步的读写锁操作。
- `estimatedTotalOperations`：记录了该 `DelayedOperationPurgatory` 中的 `DelayedOperation` 个数。
- `expirationReaper`：此字段是一个 `ShutdownableThread` 线程对象，主要有两个功能，一是推进时间轮表针，二是定期清理 `watchersForKey` 中已完成的 `DelayedOperation`，清理条件由 `purgeInterval` 字段指定。在 `DelayedOperationPurgatory` 初始化时会启动此线程。此线程的 `doWork()` 方法的代码如下：

```

override def doWork() {
    advanceClock(200L) // 此方法最长阻塞 200ms
}

// 下面是 DelayedOperationPurgatory.advanceClock() 方法的实现
def advanceClock(timeoutMs: Long) {
    timeoutTimer.advanceClock(timeoutMs) // 尝试推进时间轮的表针

    // DelayedOperation 到期后被 SystemTimer.taskExecutor 完成后，并不会通知

```



```
// DelayedOperationPurgatory 删除 DelayedOperation
// 当 DelayedOperationPurgatory 与 SystemTimer 中的 DelayedOperation 数量相差到
// 一个阈值时，会调用 purgeCompleted() 方法执行清理工作
if (estimatedTotalOperations.get - delayed > purgeInterval) {
    estimatedTotalOperations.getAndSet(delayed) // 更新 estimatedTotalOperations
    // 调用 Watchers.purgeCompleted() 方法清理已完成的 DelayedOperation
    val purged = allWatchers.map(_.purgeCompleted()).sum
}
}
```

DelayedOperationPurgatory 的核心方法有两个：一个是 checkAndComplete() 方法，主要是根据传入的 key 尝试执行对应的 Watchers 中的 DelayedOperation，通过调用 Watchers.tryCompleteWatched() 方法实现，不再赘述。另一个是 tryCompleteElseWatch() 方法，主要功能是检测 DelayedOperation 是否已经完成，若未完成则添加到 watchersForKey 以及 SystemTimer 中。具体的执行步骤如下：

```
def tryCompleteElseWatch(operation: T, watchKeys: Seq[Any]): Boolean = {
    // 步骤 1: 调用 DelayedOperation.tryComplete() 方法，尝试完成延迟操作
    var isCompletedByMe = operation.synchronized { operation.tryComplete() }
    if (isCompletedByMe) // 已完成，直接返回
    * return true

    var watchCreated = false
    // 步骤 2: 传入的 key 可能有多个，每个 key 表示一个 DelayedOperation 关心的条件
    // 将 DelayedOperation 添加到所有 key 对应的 Watchers 中
    for (key <- watchKeys) {
        // 添加过程中若已经被其他线程完成，则放弃后续添加过程，ExpiredOperationReaper 线
        // 程会定期清理 watchersForKey，所以这里不需要清理之前添加的 key
        if (operation.isCompleted())
            return false
        // 将 DelayedOperation 添加到 watchersForKey 中对应的 Watchers 中
        watchForOperation(key, operation)
        if (!watchCreated) {
            watchCreated = true
            // 增加 estimatedTotalOperations 的值
            estimatedTotalOperations.incrementAndGet()
        }
    }
}
```

```

// 步骤 3: 第二次尝试完成此 DelayedOperation, 如果成功执行, 则直接返回
isCompletedByMe = operation.synchronized { operation.tryComplete() }
if (isCompletedByMe)
    return true

// 执行到这里可以保证, 此 DelayedOperation 不会错过任何 key 上触发的 checkAndComplete()
// 步骤 4: 将 DelayedOperation 提交到 SystemTimer
if (!operation.isCompleted()) {
    timeoutTimer.add(operation)
    if (operation.isCompleted()) { // 再次检测完成情况
        operation.cancel() // 如果已完成, 则将其从 SystemTimer 中删除
    }
}
false
}

```

有读者可能会问, 为什么一个 DelayedOperation 可能会关心多个 key 呢? 这里我们还是以 DelayedProduce 为例分析: 在第 2 章介绍 ProducerRequest 时提到过, ProducerRequest 中可以包含发送往同一节点的不同 TopicAndPartition 的消息, 那么处理 ProducerRequest 时就涉及向多个分区中追加消息, 所以需要关注多个 TopicPartitionOperationKey, 只有所有的分区都满足条件才能对客户端响应。

4.4.5 DelayedProduce

经过前面的介绍, 我们已经了解了 SystemTimer 和 DelayedOperationPurgatory 的工作原理。在详细介绍 DelayedProduce 的相关实现之前, 先来了解一下当 ProducerRequest 的 acks 字段为 -1 时, 服务端的处理流程: 在 KafkaApis 中处理 ProducerRequest 的方法是 handleProducerRequest() 方法, 它会调用 ReplicaManager.appendMessages() 方法将消息追加到 Log 中, 生成相应的 DelayedProduce 对象并添加到 delayedProducePurgatory 处理。delayedProducePurgatory 是 ReplicaManager 中的字段, 它是专门用来处理 DelayedProduce 的 DelayedOperationPurgatory 对象, 其定义如下:

```

val delayedProducePurgatory = DelayedOperationPurgatory[DelayedProduce](
    purgatoryName="Produce",
    config.brokerId, config.producerPurgatoryPurgeIntervalRequests)

```

这里简略了解一下 ReplicaManager.appendMessages() 方法中与 DelayedProduce 处理相

关的部分代码：

```
def appendMessages(...) {
    // 将消息追加到 Log 中，同时还会检测 delayedFetchPurgatory 中相关 key 对应的
    // DelayedFetch，满足条件则将其执行完成（关于 delayedFetchPurgatory 和
    // DelayFetch 的内容在下一小节介绍）
    val localProduceResults = appendToLocalLog(internalTopicsAllowed,
        messagesPerPartition, requiredAcks)
    ... ..
    // 对追加结果进行转换，注意 ProducePartitionStatus 的参数
    val produceStatus = localProduceResults.map {
        case (topicPartition, result) => topicPartition ->
            ProducePartitionStatus(result.info.lastOffset + 1, new PartitionResponse(...))
    }
    ... ..
    // 下面检测是否生成 DelayedProduce，其中一个条件就是检测 ProduceRequest 中的 acks
    // 字段是否为 -1
    if (delayedRequestRequired(...)) {
        val produceMetadata = ProduceMetadata(requiredAcks, produceStatus)
        // 创建 DelayedProduce 对象
        val delayedProduce = new DelayedProduce(timeout, produceMetadata,
            this, responseCallback)
        // 将 ProduceRequest 中
        val producerRequestKeys = messagesPerPartition.keys.map(
            new TopicPartitionOperationKey(_)).toSeq

        // 尝试完成 DelayedProduce，否则将 DelayedProduce 添加到 delayedProducePurgatory 中管理
        delayedProducePurgatory.tryCompleteElseWatch(delayedProduce,
            producerRequestKeys)
    }
    ... ..
}
```

现在我们回到对 DelayedProduce 的分析，其中各个字段的含义和功能如下所述。

- delayMs: DelayedProduce 的延迟时长。
- produceMetadata: ProduceMetadata 对象。ProduceMetadata 中为一个 ProducerRequest 中的所有相关分区记录了一些追加消息后的返回结果，主要用于判断 DelayedProduce

是否满足执行条件:

```
case class ProduceMetadata(produceRequiredAcks: Short,
  produceStatus: Map[TopicPartition, ProducePartitionStatus]) {...}
```

其中, `produceRequiredAcks` 字段记录了 `ProduceRequest` 中 `acks` 字段的值, `produceStatus` 记录了每个 `Partition` 的 `ProducePartitionStatus`。 `ProducePartitionStatus` 的定义如下:

```
case class ProducePartitionStatus(requiredOffset: Long,
  responseStatus: PartitionResponse) {
  @volatile var acksPending = false
  // 省略 toString() 方法
}
```

从上面的 `ReplicaManager.appendMessages()` 的代码可以看出, `requiredOffset` 是 `ProducerRequest` 中追加到此分区的最后一个消息的 `offset`, 它会参与判断 `DelayedProduce` 是否符合执行条件, 在 `DelayedProduce.tryComplete()` 方法中介绍。 `acksPending` 字段表示是否正在等待 ISR 集合中其他副本与 Leader 副本同步 `requiredOffset` 之前的消息, 如果 ISR 集合中所有副本已经完成了 `requiredOffset` 之前消息的同步, 则此值被设置为 `false`。 `responseStatus` 字段主要用来记录 `ProducerResponse` 中的错误码。

- `responseCallback`: 任务满足条件或到期执行时, 在 `DelayedProduce.onComplete()` 方法中调用的回调函数。其主要功能是向 `RequestChannels` 中对应的 `responseQueue` 添加 `ProducerResponse`, 之后 `Processor` 线程会将其发送给客户端。
- `replicaManager`: 此 `DelayedProduce` 关联的 `ReplicaManager` 对象。

在 `DelayedProduce` 初始化时, 首先会对 `produceMetadata` 字段中的 `produceStatus` 集合进行设置。初始化的代码如下:

```
// 根据前面写入消息返回的结果, 设置 ProducePartitionStatus 的 acksPending 字段和
// responseStatus 字段的值
produceMetadata.produceStatus.foreach {
  case (topicPartition, status) =>
    // 对应分区的写入操作成功, 则等待 ISR 集合中的副本完成同步
    // 如果写入操作出现异常, 则该分区不需要等待
    if (status.responseStatus.errorCode == Errors.NONE.code) {
```



```

        status.acksPending = true
    // 下面是预设错误码, 如果 ISR 集合中的副本在此请求超时之前顺利完成了同步, 会清除此错误码
        status.responseStatus.errorCode = Errors.REQUEST_TIMED_OUT.code
    } else {
        // 如果追加日志时已经抛出异常, 则不必等待此 Partition 对应的 ISR 返回 ACK 了
        status.acksPending = false
    }
}

```

DelayedProduce 实现了 DelayedOperation.tryComplete() 方法, 其主要逻辑是检测是否满足 DelayedProduce 的执行条件, 并在满足执行条件时调用 forceComplete() 方法完成该延迟任务。满足下列任一条件, 即表示此分区已经满足 DelayedProduce 的执行条件。只有 ProducerRequest 中涉及的所有分区都满足条件, DelayedProduce 才能最终执行。

(1) 该分区出现了 Leader 副本的迁移。该分区的 Leader 副本不再位于此节点上, 此时会更新对应 ProducePartitionStatus 中记录的错误码。

(2) 正常情况下, ISR 集合中所有副本都完成了同步后, 该分区的 Leader 副本的 HW 位置已经大于对应的 ProduceStatus.requiredOffset。此时会清空初始化中设置的超时错误码。

(3) 如果出现异常, 则更新分区对应的 ProducePartitionStatus 中记录的错误码。

DelayedProduce.tryComplete() 方法的实现如下:

```

override def tryComplete(): Boolean = {
    // 遍历 produceMetadata 中的所有分区的状态
    produceMetadata.produceStatus.foreach {
        case (topicAndPartition, status) =>
            if (status.acksPending) { // 检查此分区是否已经满足 DelayedProduce 执行条件
                // 获取对应的 Partition 对象
                val partitionOpt = replicaManager.getPartition(topicAndPartition.
topic,
                    topicAndPartition.partition)
                val (hasEnough, errorCode) = partitionOpt match {
                    case Some(partition) =>
                        // 检查此分区的 HW 位置是否大于 requiredOffset。这里涉及 Partition 类中的

```

```
// checkEnoughReplicasReachOffset() 方法, 此方法会在后面介绍 Partition 时详细分析
    partition.checkEnoughReplicasReachOffset(status.requiredOffset)
    case None =>
        // 条件 1: 找不到此分区的 Leader
        (false, Errors.UNKNOWN_TOPIC_OR_PARTITION.code)
    }

    if (errorCode != Errors.NONE.code) { // 条件 3: 出现异常
        status.acksPending = false
        status.responseStatus.errorCode = errorCode
        // 条件 2: 此分区 Leader 副本的 HW 大于对应的 requiredOffset
    } else if (hasEnough) {
        status.acksPending = false
        status.responseStatus.errorCode = Errors.NONE.code
    }
}

// 检查全部的分区是否都已经符合 DelayedProduce 的执行条件
if (!produceMetadata.produceStatus.values.exists(p => p.acksPending))
    forceComplete()
else
    false
}
```

通过前面的分析, `onComplete()` 方法才是 `DelayedProduce` 执行的真正逻辑, 其代码如下:

```
override def onComplete() {
    // 根据 ProduceMetadata 记录的相关信息, 为每个 Partition 产生响应状态
    val responseStatus = produceMetadata.produceStatus.mapValues(
        status => status.responseStatus)
    responseCallback(responseStatus) // 调用 responseCallback 回调函数
}
```

请读者注意一个细节, 如果 `DelayedProduce` 是到期执行, 则返回的错误码是在其初始化过程中预先设置的 `Errors.REQUEST_TIMED_OUT.code`。

最后, 来看一下 `responseCallback` 这个回调函数的具体实现, 它会向 `RequestChannels` 中对应的 `responseQueue` 队列添加 `ProducerResponse`, 最终 `Processor` 线程会将

ProducerResponse 返回给生产者。读者可以跟踪一下代码，发现此回调函数是在 KafkaApis.handleProducerRequest() 方法中定义 sendResponseCallback() 函数，其具体代码如下：

```
def sendResponseCallback(responseStatus: Map[TopicPartition, PartitionResponse])
{
    // 生成响应状态集合，其中包括通过授权验证并处理完成的状态（responseStatus），
    // 以及未通过授权验证的状态
    val mergedResponseStatus = responseStatus ++ unauthorizedRequestInfo.
mapValues(_ =>
    new PartitionResponse(Errors.TOPIC_AUTHORIZATION_FAILED.code,
        -1, Message.NoTimestamp))

    var errorInResponse = false // 标识处理 ProducerRequest 的过程中是否出现异常
    mergedResponseStatus.foreach {
        case (topicPartition, status) =>
            if (status.errorCode != Errors.NONE.code) {
                errorInResponse = true
            }
    }

    // 定义 produceResponseCallback() 回调函数
    def produceResponseCallback(delayTimeMs: Int) {
        // 处理 acks 字段为 0 的情况，即生产者不需要服务端返回响应
        if (produceRequest.acks == 0) {
            if (errorInResponse) {
                // 处理 ProducerRequest 过程中出现异常，则向对应 responseQueue 中
                // 添加 RequestChannel.CloseConnectionAction 类型响应，关闭连接
                requestChannel.closeConnection(request.processor, request)
            } else {
                // 未出现异常，向对应 responseQueue 中添加 NoOpAction 类型响应，继续读取客户端的请求
                requestChannel.noOperation(request.processor, request)
            }
        } else { // 处理 acks 字段为 1 或 -1 的情况，即生产者需要服务端响应
            // 创建消息头
            val respHeader = new ResponseHeader(request.header.correlationId)
            val respBody = request.header.apiVersion match { // 创建消息体
```



```

    ... ..
}
// 向对应 responseQueue 中添加 SendAction 类型响应, 将响应返回给客户端
requestChannel.sendResponse(new RequestChannel.Response(request,
    new ResponseSend(request.connectionId, respHeader, respBody)))
}
}

// ClientQuotaManager 是用来记录监控数据的, 我们这里不做详细介绍, 在其中会调用
// produceResponseCallback 这个回调函数
quotaManagers(ApiKeys.PRODUCE.id).recordAndMaybeThrottle(request.header.
clientId,
    numBytesAppended, produceResponseCallback)
}

// 下面是 ClientQuotaManager.recordAndMaybeThrottle() 方法的实现
def recordAndMaybeThrottle(clientId:String, value: Int, callback: Int =>
Unit):Int = {
    ... ..
    var throttleTimeMs = 0
    try {
        callback(0) // 调用 callback, 即上面介绍的 produceResponseCallback
    } catch {
        // 异常处理 (略)
    }
    throttleTimeMs
}

```

整个 `ProducerRequest` 以及 `DelayedProduce` 相关的处理流程到这里就已经介绍完了。为了让读者更好地理解这个流程, 我们进行简单总结, 并给出图 4-37, 读者可以结合前面的代码分析与此图深入理解。

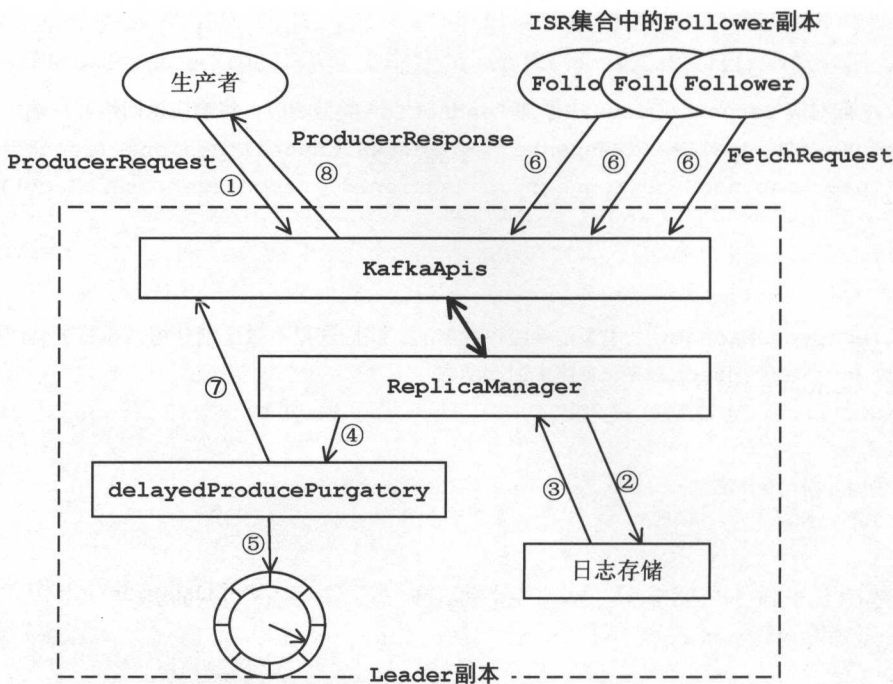


图 4-37

(1) 生产者发送 `ProducerRequest` 向某些指定分区追加消息。

(2) `ProducerRequest` 经过网络层和 API 层的处理到达 `ReplicaManager`，它会将消息交给日志存储子系统进行处理，最终追加到对应的 Log 中。同时还会检测 `delayedFetchPurgatory` 中相关 key 对应的 `DelayedFetch`，满足条件则将其执行完成，这部分内容在下一节中介绍。

(3) 日志存储子系统返回追加消息的结果。

(4) `ReplicaManager` 为 `ProducerRequest` 生成 `DelayedProduce` 对象，并交由 `delayedProducePurgatory` 管理。

(5) `delayedProducePurgatory` 使用 `SystemTimer` 管理 `DelayedProduce` 是否超时。

(6) ISR 集合中的 Follower 副本发送 `FetchRequest` 请求与 Leader 副本同步消息。同时，也会检查 `DelayedProduce` 是否符合执行条件。

(7) `DelayedProduce` 执行时会调用回调函数产生 `ProducerResponse`，并将其添加到 `RequestChannels` 中。

(8) 由网络层将 `ProducerResponse` 返回给客户端。

4.4.6 DelayedFetch

`DelayedFetch` 是 `FetchRequest` 对应的延迟操作，它的原理与 `DelayedProduce` 类似。我们先来粗略分析 `FetchRequest` 的处理流程：来自消费者或 Follower 副本的 `FetchRequest` 由 `KafkaApis.handleFetchRequest()` 方法处理，它会调用 `ReplicaManager.fetchMessages()` 方法从相应的 Log 中读取消息，并生成 `DelayedFetch` 添加到 `delayedFetchPurgatory` 中处理。`delayedFetchPurgatory` 是 `ReplicaManager` 中的字段，它是专门用来处理 `DelayedFetch` 的 `DelayedOperationPurgatory` 对象。其定义如下：

```
val delayedFetchPurgatory = DelayedOperationPurgatory[DelayedFetch](
  _ purgatoryName = "Fetch",
  _ _ config.brokerId, config.fetchPurgatoryPurgeIntervalRequests)
```

这里简略分析一下 `ReplicaManager.fetchMessages()` 方法中与 `DelayedFetch` 相关的代码：

```
def fetchMessages(...) {
  // 只有 Follower 副本才有 replicaId，而消费者是没有的，消费者的 replicaId 始终是 -1
  val isFromFollower = replicaId >= 0
  // 从 Log 中读取消息
  val logReadResults = readFromLocalLog(fetchOnlyFromLeader,
  _ _ fetchOnlyCommitted, fetchInfo)
  // 检测 FetchRequest 是否来自 Follower 副本
  if (Request.isValidBrokerId(replicaId)) {
    // updateFollowerLogReadResults() 方法用来处理来自 Follower 副本的
    // FetchRequest 请求，主要做下面 4 件事：
    // (1) 在 Leader 中维护了 Follower 副本的各种状态，这里会更新对应 Follower 副本的
    // 状态，例如，更新 LEO、更新 lastCaughtUpTimeMsUnderlying 等
    // (2) 检测是否需要将 ISR 集合进行扩张，如果 ISR 集合发生变化，则将新的 ISR 集合的记
    // 录保存到 ZooKeeper 中
    // (3) 检测是否后移 Leader 的 HW
    // (4) 检测 delayedProducePurgatory 中相关 key 对应的 DelayedProduce，满足条件
    // 则将其执行完成
    updateFollowerLogReadResults(replicaId, logReadResults)
```

```

    // updateFollowerLogReadResults() 方法的具体实现在后面会详细介绍
}

// 统计从 Log 中读取的字节总数
val bytesReadable = logReadResults.values.map(_.info.messageSet.
sizeInBytes).sum
// 统计在从 Log 中读取消息的时候, 是否发生了异常
val errorReadingData = logReadResults.values.foldLeft(false)
((errorIncurred,
    readResult) =>    errorIncurred || (readResult.errorCode != Errors.
NONE.code))

// 判断是否能够立即返回 FetchResponse, 下面四个条件满足任意一个就可以立即返回
// FetchResponse:
// (1) FetchRequest 的 timeout<=0, 即消费者或 Follower 副本不希望等待
// (2) FetchRequest 没有指定要读取的分区, 即 fetchInfo.size <= 0
// (3) 已经读取了足够的数据, 即 bytesReadable >= fetchMinBytes
// (4) 在读取数据的过程中发生了异常, 即检查 errorReadingData
if (timeout <= 0 || fetchInfo.size <= 0 || bytesReadable >= fetchMinBytes
    || errorReadingData) {
    ... ..
    // 直接调用回调函数, 生成并发送 FetchResponse
    responseCallback(fetchPartitionData)
} else {
    val fetchPartitionStatus = logReadResults.map { // 对读取 Log 的结果进行转换
        case (topicAndPartition, result) =>
            (topicAndPartition, FetchPartitionStatus(result.info.fetchOffsetMetadata,
                fetchInfo.get(topicAndPartition).get))
    }

    val fetchMetadata = FetchMetadata(fetchMinBytes, fetchOnlyFromLeader,
        fetchOnlyCommitted, isFromFollower, fetchPartitionStatus)
    // 创建 DelayedFetch 对象
    val delayedFetch = new DelayedFetch(timeout, fetchMetadata, this,
responseCallback)
    // 创建 delayedFetchKeys
    val delayedFetchKeys = fetchPartitionStatus.keys.map(

```

```

new TopicPartitionOperationKey(_)).toSeq

// 尝试完成 DelayedFetch, 否则将 DelayedFetch 添加到 delayedFetchPurgatory 中管理
delayedFetchPurgatory.tryCompleteElseWatch(delayedFetch, delayedFetchKeys)
}
}

```

我们大致上可以了解到 DelayedProduce 和 DelayedFetch 之间的关联：在处理 ProducerRequest 的过程中会向 Log 中添加数据，可能会后移 Leader 副本的 LEO，Follower 副本就可以读取到足量的数据，所以会尝试完成 DelayedFetch；在处理来自 Follower 副本的 FetchRequest 过程中，可能会后移 HW，所以会尝试完成 DelayedProduce，这样两者可以很好地协同工作了。

现在回到对 DelayedFetch 的分析，其中字段的含义如下所述。

- delayMs：延迟操作的延迟时长。
- fetchMetadata：FetchMetadata 对象。FetchMetadata 中为 FetchRequest 中的所有相关分区记录了相关状态，主要用于判断 DelayedProduce 是否满足执行条件。

```

case class FetchMetadata(fetchMinBytes: Int, fetchOnlyLeader: Boolean,
    fetchOnlyCommitted: Boolean, isFromFollower: Boolean,
    fetchPartitionStatus: Map[TopicAndPartition, FetchPartitionStatus])
{
    ..... // 省略 toString() 方法
}

```

其中，fetchMinBytes 字段记录了需要读取的最小字节数，fetchPartitionStatus 记录了每个分区的 FetchPartitionStatus。FetchPartitionStatus 的代码如下：

```

case class FetchPartitionStatus(startOffsetMetadata: LogOffsetMetadata,
    fetchInfo: PartitionFetchInfo) {
    ..... // 省略 toString() 方法
}

```

其中，startOffsetMetadata 记录了在前面读取 Log 时已经读取到的 offset 位置。fetchInfo 记录 FetchRequest 携带的一些信息，主要是请求的 offset 以及读取最大字节数。

- responseCallback：任务满足条件或到期执行时，在 DelayedFetch.onComplete() 方

法中调用的回调函数，其主要功能是创建 `FetchResponse` 并添加到 `RequestChannels` 中对应的 `responseQueue` 队列中。

`DelayedFetch.tryComplete()` 方法主要负责检测是否满足 `DelayedFetch` 的执行条件，并在满足条件时调用 `forceComplete()` 方法执行延迟操作。满足下面任一条件，即表示此分区满足 `DelayedFetch` 的执行条件：

- (1) 发生 Leader 副本迁移，当前节点不再是该分区的 Leader 副本所在的节点。
- (2) 当前 Broker 找不到需要读取数据的分区副本。
- (3) 开始读取的 offset 不在 `activeSegment` 中，此时可能是发生了 Log 截断，也有可能发生了 roll 操作产生了新的 `activeSegment`。
- (4) 累计读取的字节数超过最小字节数限制。

与 `DelayedProduce` 不同，`DelayedFetch` 中只要有一个 Partition 满足任一执行条件，`DelayedFetch` 就会最终执行。下面是 `tryComplete()` 方法的代码：

```
override def tryComplete() : Boolean = {
  var accumulatedSize = 0
  // 遍历 fetchMetadata 中的所有 Partition 的状态
  fetchMetadata.fetchPartitionStatus.foreach {
    case (topicAndPartition, fetchStatus) =>
      // 获取前面读取 Log 时的结束位置
      val fetchOffset = fetchStatus.startOffsetMetadata
      try {
        if (fetchOffset != LogOffsetMetadata.UnknownOffsetMetadata) {
          // 查找分区的 Leader 副本，如果找不到就会抛出异常
          val replica = replicaManager.getLeaderReplicaIfLocal(topicAndPartition.topic, topicAndPartition.partition)

          // 根据 FetchRequest 请求的来源设置能读取的最大 offset 值。很显然，消费者对
          // 应的 endOffset 是 HW，而 Follower 副本对应的 endOffset 是 LEO
          val endOffset =
            if (fetchMetadata.fetchOnlyCommitted)
              replica.highWatermark
            else
```

```

        replica.logEndOffset

        // 检查上次读取后 endOffset 是否发生变化。如果没改变，之前读不到足够的数据现
        // 在还是读不到，即任务条件依然不满足；如果变了，则继续下面的检查，看是否真
        // 正满足任务执行条件
        if (endOffset.messageOffset != fetchOffset.messageOffset) {
            if (endOffset.onOlderSegment(fetchOffset)) { // 条件 3
                // 此时，endOffset 出现减小的情况，跑到 baseOffset 较小的 Segment 上了，
                // 可能是 Leader 副本的 Log 出现了 truncate 操作，如图 4-38 所示
                return forceComplete()
            } else if (fetchOffset.onOlderSegment(endOffset)) { // 条件 3
                // 此时，fetchOffset 虽然依然在 endOffset 之前，但是产生了新的 activeSegment
                // fetchOffset 在较旧的 LogSegment，而 endOffset 在 activeSegment
                return forceComplete()
            } else if (fetchOffset.messageOffset < endOffset.messageOffset)
        }
        // endOffset 和 fetchOffset 依然在同一个 LogSegment 中，且
        // endOffset 向后移动，那就尝试计算累计的字节数
        accumulatedSize += math.min(endOffset.positionDiff(fetchOffset),
            fetchStatus.fetchInfo.fetchSize)
    }
}
} catch { // 条件 1 和条件 2
    // 异常处理（略）
    return forceComplete()
}

// 累计读取字节数足够，满足条件 4
if (accumulatedSize >= fetchMetadata.fetchMinBytes)
    forceComplete() // 调用 onComplete() 方法，在其中会重新读取数据
else
    false
}

```

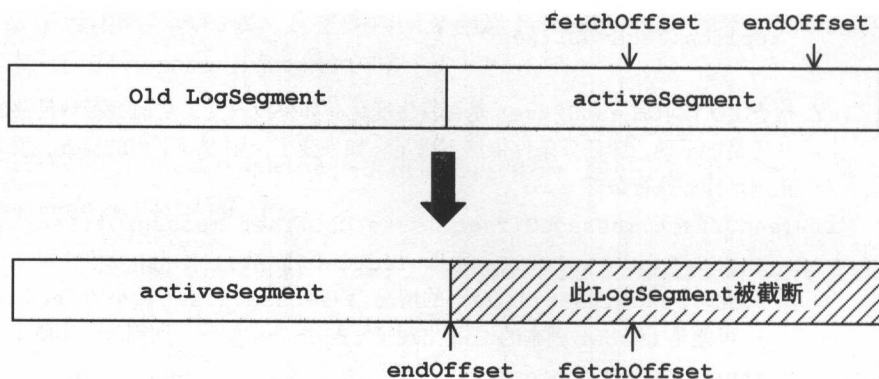


图 4-38

分析到这里读者可能会问，为什么产生新的 activeSegment 时，就不用管 fetchMinBytes 这个最小读取字节数的限制而直接返回 FetchResponse 呢？请读者回顾前面对 LogSegment.read() 方法的介绍，read() 方法返回的是分片的 FileMessageSet 对象，而并没有真正从文件中读取数据到内存中。如果这里的 fetchOffset 与 endOffset 分属不同 FileMessageSet，read() 方法的结果就无法使用一个分片的 FileMessageSet 对象表示了。读者可能又会问，LogSegment.read() 方法的返回值为什么要设计成返回分片的 FileMessage 对象呢？这么设计的主要目的是为了使用“零拷贝（Zero Copy）”技术。

零拷贝

简单说，在消费者获取消息时，服务器先从硬盘读出数据到内存，然后将内存中的数据原封不动地通过 Socket 发送给消费者。虽然这个操作描述非常简单，但是其中涉及的步骤非常多，效率也比较差，尤其是当数据量较大时。按照这种设计，其底层执行步骤大致如下：首先，应用程序调用 read() 方法时需要从用户态切换到内核态，将数据从磁盘上读取出来保存到内核缓冲区中；然后，内核缓冲区中的数据传输到应用程序，此时 read() 方法调用结束，从内核态切换到用户态；之后，应用程序执行 send() 方法，需要从用户态切换到内核态，将数据传输给 Socket Buffer；最后，内核会将 Socket Buffer 中的数据发送 NIC Buffer（网卡缓冲区）进行发送，此时 send() 方法结束，从内核态切换到用户态。如图 4-39 所示，在这个过程中涉及四次上下文切换（Context switch）以及四次数据复制，并且其中有两次复制操作由 CPU 完成。但是在这个过程中，数据完成没有进行变化，仅仅是从磁盘复制到了网卡缓冲区中，会浪费大量的 CPU 周期。

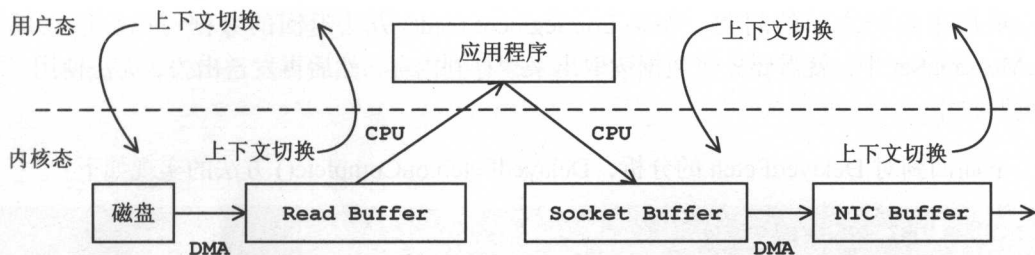


图 4-39

通过“零拷贝”技术可以去掉这些无谓的数据复制操作，同时也会减少上下文切换的次数。大致步骤如下：首先，应用程序调用 `transferTo()` 方法，DMA 会将文件数据发送到内核缓冲区；然后，Socket Buffer 追加数据的描述信息；最后，DMA 将内核缓冲区的数据发送到网卡缓冲区，这样就完全解放了 CPU，实现了零拷贝，如图 4-40 所示。

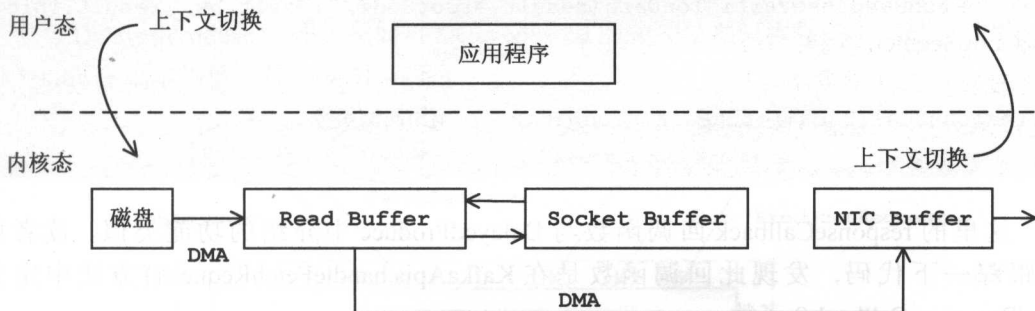


图 4-40

我们回到 `LogSegment.writeTo()` 方法，其中与“零拷贝”相关的代码如下：

```
def writeTo(destChannel: GatheringByteChannel, writePosition: Long, size:
Int):Int = {
    ... ..// 边界检查(略)
    val position = start + writePosition // 计算起始位置
    val count = math.min(size, sizeInBytes) // 计算发送字节数
    val bytesTransferred = (destChannel match {
        // 在底层调用的是fileChannel.transferTo()方法，实现“零拷贝”
        case tl: TransportLayer => tl.transferFrom(channel, position, count)
        case dc => channel.transferTo(position, count, dc)
    }).toInt
    bytesTransferred
}
```


最后来解释之前的问题，如果 `LogSegment.read()` 方法返回的数据可以分散在两个 `FileMessageSet` 中，就需要先将数据读取出来缓存到内存，然后再发送出去，无法使用“零拷贝”技术。

下面回到对 `DelayedFetch` 的分析，`DelayedFetch.onComplete()` 方法的实现如下：

```
override def onComplete() {
    // 重新从 Log 中读取数据
    val logReadResults = replicaManager.readFromLocalLog(fetchMetadata.
fetchOnlyLeader,
    fetchMetadata.fetchOnlyCommitted,
    fetchMetadata.fetchPartitionStatus.mapValues(status => status.fetchInfo))
    // 将读取结果进行封装
    val fetchPartitionData = logReadResults.mapValues(result =>
        FetchResponsePartitionData(result.errorCode, result.hw, result.info.
messageSet))

    responseCallback(fetchPartitionData) // 调用回调函数
}
```

这里的 `responseCallback` 回调函数与 `DelayedProduce` 中介绍的功能类似。读者可以跟踪一下代码，发现此回调函数是在 `KafkaApis.handleFetchRequest()` 方法中定义 `sendResponseCallback()` 函数。

```
def sendResponseCallback(responsePartitionData:
    Map[TopicAndPartition, FetchResponsePartitionData]) {
    val convertedPartitionData = ... ..// 协议的版本转换（略）
    // 将之前把未认证通过的集合与 convertedPartitionData 合并
    val mergedPartitionData = convertedPartitionData ++ unauthorizedPartitionData
    .....// 统计数据供监控使用（略）

    // 定义 fetchResponseCallback() 函数
    def fetchResponseCallback(delayTimeMs: Int) {
        // 生成 FetchResponse 对象
        val response = FetchResponse(fetchRequest.correlationId,
            mergedPartitionData, fetchRequest.versionId, delayTimeMs)

        // 向对应 responseQueue 中添加一个 SendAction 的 Response，其中封装了
```

```

// 上面的 FetchResponse 对象
requestChannel.sendResponse(new RequestChannel.Response(request,
    new FetchResponseSend(request.connectionId, response)))
}
if (fetchRequest.isFromFollower) {
    // 调用 fetchResponseCallback() 返回 FetchResponse
    fetchResponseCallback(0)
} else {
    // 底层也是调用 fetchResponseCallback()
    quotaManagers(ApiKeys.FETCH.id).recordAndMaybeThrottle(... ..
        fetchResponseCallback)
}
}
}

```

与 DelayedProduce 一样，我们对 DelayedFetch 的相关处理流程做一下总结，这里以来自 Follower 副本为例，如图 4-41 所示。

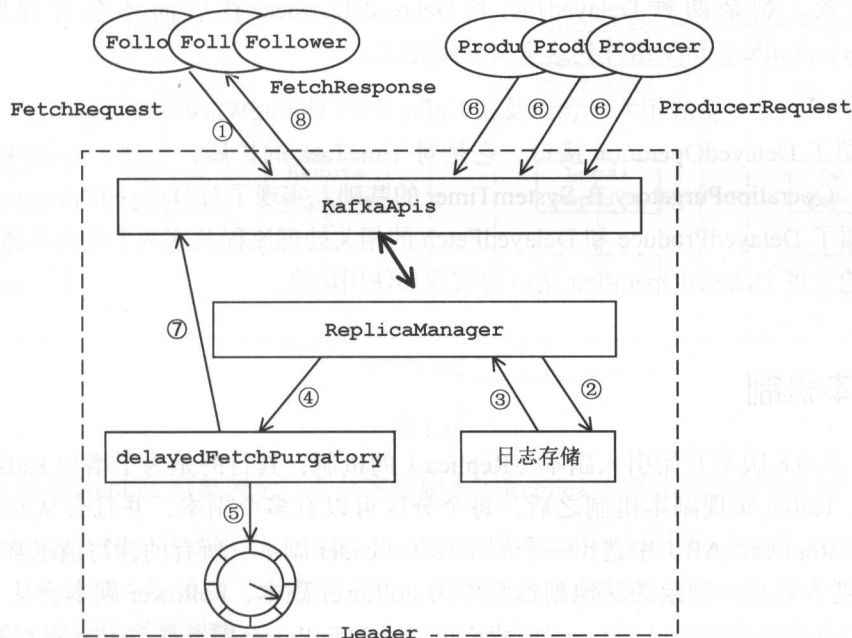


图 4-41

(1) Follower 副本发送 FetchRequest，从某些分区中获取消息。

(2) `FetchRequest` 经过网络层和 API 层的处理, 到达 `ReplicaManager`, 它会从日志存储子系统中读取数据, 并检测是否要更新 ISR 集合、HW 等, 之后还会执行 `delayedProducePurgatory` 中满足条件的相关 `DelayedProduce`。

(3) 日志存储子系统返回读取消息以及相关信息, 例如此次读取到的 offset 等。

(4) `ReplicaManager` 为 `FetchRequest` 生成 `DelayedFetch` 对象, 并交由 `delayedFetchPurgatory` 管理。

(5) `delayedFetchPurgatory` 使用 `SystemTimer` 管理 `DelayedFetch` 是否超时。

(6) 生产者发送 `ProducerRequest` 请求追加消息。同时也会检查 `DelayedFetch` 是否符合执行条件。

(7) `DelayedFetch` 执行时会调用回调函数产生 `FetchResponse`, 添加到 `RequestChannels` 中。

(8) 由网络层将 `FetchResponse` 返回给客户端。

这里主要介绍了 `DelayProduce` 和 `DelayFetch` 两种 `DelayedOperation` 实现细节和应用场景, 剩余两种 `DelayedJoin` 和 `DelayedHeartbeat` 在后面还会有详细介绍。`DelayedOperationPurgatory` 组件到这里就介绍完了。

本节介绍了时间轮的相关概念以及在 Kafka 中的 `TimingWheel`、`SystemTimer` 的实现。之后, 介绍了 `DelayedOperation` 接口, 它是对 `TimeTask` 的扩展, 实现了有条件的延迟操作。`DelayedOperationPurgatory` 在 `SystemTimer` 的基础上实现了对 `DelayedOperation` 的管理。最后, 介绍了 `DelayedProduce` 和 `DelayedFetch` 的相关处理流程及这两个类的具体实现, 让读者清楚地了解 `DelayedOperation` 接口的实现和使用场景。

4.5 副本机制

Kafka 从 0.8 版本开始引入副本 (Replica) 的机制, 其目的是为了增加 Kafka 集群的高可用性。Kafka 实现副本机制之后, 每个分区可以有多个副本, 并且会从其副本集合 (Assigned Replica, AR) 中选出一个副本作为 Leader 副本, 所有的读写请求都由选举出的 Leader 副本处理。剩余的其他副本都作为 Follower 副本, Follower 副本会从 Leader 副本处获取消息并更新到自己的 Log 中。我们可以认为 Follower 副本是 Leader 副本的热备份。一般情况下, 同一分区的多个副本会被均匀地分配到集群中的不同 Broker 上, 当 Leader 副本的所在的 Broker 出现故障后, 可以重新选举新的 Leader 副本继续对外提供服务。通过这种方式提高了 Kafka 集群的可用性。

如果一个分区的全部副本被分配到同一个 Broker 上, 那么该 Broker 故障就会导致整个分区不可用; 如果多个副本集中分配到了某个 Broker 上, 则会出现 Broker 负载不均衡的情况。这两种情况都不是我们希望看到的, 所以均匀地分配副本是一项很重要的工作。分配副本的具体实现在第 5 章介绍 `kafka-topics` 脚本时会详细介绍。

在第 1 章, 已经介绍过了 ISR 集合、HW、LEO 等概念, 这里就不再赘述了, 建议读者先回顾这几个概念, 再开始下面的分析。

4.5.1 副本

在一个分区的 Leader 副本中会维护自身以及所有 Follower 副本的相关状态, 而 Follower 副本只维护自己的状态。此外, 还有“本地副本”和“远程副本”两个概念需要读者注意, “本地副本”是指副本对应的 Log 分配在当前的 Broker 上, “远程副本”则是指副本对应的 Log 分配在其他的 Broker 上, 在当前 Broker 上仅仅维护了副本的 LEO 等信息。一个副本是“本地副本”还是“远程副本”与它是 Leader 副本还是 Follower 副本没有直接联系, 如图 4-42 所示。

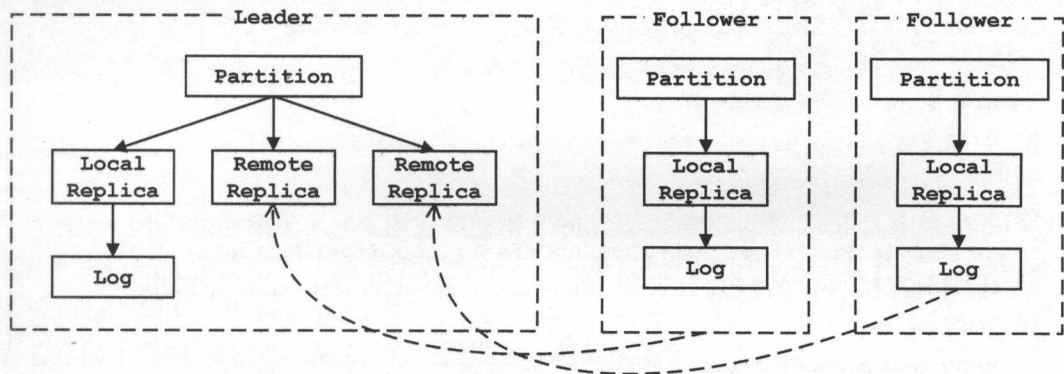


图 4-42

Kafka 使用 Replica 对象表示一个分区的副本, Replica 中重要字段的含义如下所述。

- `brokerId`: 标识该副本所在的 Broker 的 id。在后面的分析中会看到, 区分一个副本是“本地副本”还是“远程副本”, 可以通过 `Replica.brokerId` 字段与当前 Broker 的 Id 进行比较来判断。
- `highWatermarkMetadata`: `LogOffsetMetadata` 对象, 此字段使用来记录 HW (HighWatermark) 的值。消费者只能获取到 HW 之前的消息, 其后的消息对消费者是不可见的。此字段由 Leader 副本负责更新维护, 更新时机是消息被 ISR 集合

中所有副本成功同步，即消息被成功提交。LogOffsetMetadata 在前面分析 Log 类的实现时介绍过，这里不再赘述。

- logEndOffsetMetadata: LogOffsetMetadata 对象。对于本地副本，此字段记录的是追加到 Log 中的最新消息的 offset，可以直接从 Log.nextOffsetMetadata 字段中获取。对于远程副本，此字段含义相同，但是由其他 Broker 发送请求来更新此值，并不能直接从本地获取到。
- partition: Partition 对象，此副本对应的分区。
- log: 本地副本对应的 Log 对象，远程副本的此字段为空。通过此字段可以区分“本地副本”和“远程副本”。
- lastCaughtUpTimeMsUnderlying: AtomicLong 对象，用于记录 Follower 副本最后一次追赶上 Leader 的时间戳。

在 Replica 中提供了上述字段的读写方法，看一下其代码：

```
def isLocal: Boolean = {
  log match { // 创建 Replica 对象时指定了 Log，则表示是本地副本
    case Some(l) => true
    case None    => false
  }
}

private def logEndOffset_(newLogEndOffset: LogOffsetMetadata) {
  // 对于本地副本，不能直接更新 LEO，其 LEO 由 Log.logEndOffsetMetadata 字段决定
  if (isLocal) {
    throw new KafkaException("...")
  } else { // 对于远程副本，LEO 是通过请求进行更新的
    logEndOffsetMetadata = newLogEndOffset
  }
}

def logEndOffset = // 获取 LEO，本地副本和远程副本的获取方式不同
  if (isLocal) log.get.logEndOffsetMetadata
  else logEndOffsetMetadata
```

```

def updateLogReadResult(logReadResult: LogReadResult) {

    logEndOffset = logReadResult.info.fetchOffsetMetadata // 更新 LEO
    // 更新 lastCaughtUpTimeMsUnderlying, 表示此时 Follower 副本的同步进度完全追赶上
    // 了 Leader 副本
    if (logReadResult.isReadFromLogEnd) {
        lastCaughtUpTimeMsUnderlying.set(time.milliseconds)
    }
}

def highWatermark_(newHighWatermark: LogOffsetMetadata) {
    if (isLocal) { // 只有本地副本可以更新 HW
        highWatermarkMetadata = newHighWatermark
    } else {
        throw new KafkaException("...") // 远程副本更新 HW 会抛出异常
    }
}

def highWatermark = highWatermarkMetadata // 获取 HW

```

4.5.2 分区

在上一小节中介绍了 Kafka 服务端使用 Replicar 表示副本以及 Replicar 中维护的信息，其中的 partition 字段指向了副本所属的分区。服务端使用 Partition 表示分区，Partition 负责管理每个副本对应的 Replicar 对象，进行 Leader 副本的切换，ISR 集合的管理以及调用日志存储子系统完成写入消息，它还提供了一些其他的辅助方法。

Partition 中的核心字段的含义如下所述。

- topic 和 partitionId: 此 Partition 对象代表的 Topic 名称和分区编号。
- localBrokerId: 当前 Broker 的 id, 可以与 replicaId 比较, 从而判断指定的 Replicar 对是否表示本地副本。
- logManager: 当前 Broker 上的 LogManager 对象。
- zkUtils: 操作 ZooKeeper 的辅助类。
- leaderEpoch: Leader 副本的年代信息。

- `leaderReplicaIdOpt`: 该分区的 Leader 副本的 id。
- `inSyncReplicas`: `Set[Replica]` 类型, 该集合维护了该分区的 ISR 集合, ISR 集合是 AR 集合的子集。
- `assignedReplicaMap`: `Pool[Int, Replica]` 类型, 维护了该分区的全部副本的集合 (AR 集合) 的信息。

Partition 中的方法按照功能可以划分为下列五类。

- 获取 (或创建) Replica: `getOrCreateReplica()` 方法。
- 副本的 Leader/Follower 角色切换: `makeLeader()` 方法和 `makeFollower()` 方法。
- ISR 集合管理: `maybeExpandIsr()` 方法和 `maybeShrinkIsr()` 方法。
- 调用日志存储子系统完成消息写入: `appendMessagesToLeader()` 方法。
- 检测 HW 的位置: `checkEnoughReplicasReachOffset()` 方法

上述五类方法为 `ReplicaManager` 的实现提供了基础支持。其他较为简单的辅助方法不再做详细介绍, 请读者参考源码学习。

创建副本

`getOrCreateReplica()` 方法主要负责在 AR 集合 (`assignedReplicaMap`) 中查找指定副本的 Replica 对象, 如果查找不到则创建 Replica 对象并添加到 AR 集合中管理。如果创建的是 Local Replica, 还会创建 (或恢复) 对应的 Log 并初始化 (或恢复) HW。HW 与 `Log.recoveryPoint` 类似, 也会需要记录到文件中保存, 在每个 log 目录下都有一个 `replication-offset-checkpoint` 文件记录了此目录下每个分区的 HW。在 `ReplicaManager` 启动时会读取此文件到 `highWatermarkCheckpoints` 这个 Map 中, 之后会定时更新 `replication-offset-checkpoint` 文件。

```
def getOrCreateReplica(replicaId: Int = localBrokerId): Replica = {
  val replicaOpt = getReplica(replicaId) // 从 assignedReplicaMap 中获查找的
  Replica 对象
  replicaOpt match {
    case Some(replica) => replica // 查找到指定 Replica 对象, 直接返回
    case None =>
      if (isReplicaLocal(replicaId)) { // 判断是否为 Local Replica
        // 获取配置信息, ZooKeeper 中的配置会覆盖默认的配置
        val config = LogConfig.fromProps(logManager.defaultConfig.originals,
```

```

        AdminUtils.fetchEntityConfig(zkUtils, ConfigType.Topic,
topic))

// 创建 Local Replica 对应的 Log, 如果 Log 已经存在, 则直接返回。请读者参考 LogManager 小节
    val log = logManager.createLog(TopicAndPartition(topic, partitionId),
config)

    // 获取指定 log 目录对应的 OffsetCheckpoint 对象, 它负责管理该 log 目录下的
    // replication-offset-checkpoint 文件。OffsetCheckpoint 类在 LogManager 中已介绍
    val checkpoint = replicaManager.highWatermarkCheckpoints(
        log.dir.getParentFile.getAbsolutePath)

    // 将 replication-offset-checkpoint 文件中记录的 HW 信息形成 Map
    val offsetMap = checkpoint.read
    .....// 边界检查(略)

    // 根据 TopicAndPartition 找到对应的 HW, 再与 LEO 比较, 此值会作为此副本的 HW
    val offset = offsetMap.getOrElse(TopicAndPartition(topic,
partitionId),
        0L).min(log.logEndOffset)

    // 创建 Replica 对象, 并添加到 assignedReplicaMap 集合中管理
    val localReplica = new Replica(replicaId, this, time, offset,
Some(log))
    addReplicaIfNotExists(localReplica)
} else {
    // Remote Replica 比较简单, 直接创建 Replica 对象并添加到 AR 集合中即可
    val remoteReplica = new Replica(replicaId, this, time)
    addReplicaIfNotExists(remoteReplica)
}
getReplica(replicaId).get // 最后返回对应的 Replica
}
}

```

副本角色切换

Broker 会根据 KafkaController 发送的 LeaderAndISRRequest 请求控制副本的 Leader/Follower 角色切换。Partition.makeLeader() 方法是处理 LeaderAndISRRequest 中比较重要的

环节之一，它会将 Local Replica 设置成 Leader 副本，其调用栈如图 4-43 所示。

```

▼ ① Partition.makeLeader(int, PartitionState, int) (kafka.cluster)
  ▼ ① ReplicaManager.makeLeaders(int, int, Map<Partition, PartitionState>, int, Map<TopicPartition, Object>) (kafka.server)
    ▼ ① ReplicaManager.becomeLeaderOrFollower(int, LeaderAndIsrRequest, MetadataCache, Function2<Iterable<Partition>,
      ▼ ① KafkaApis.handleLeaderAndIsrRequest(Request) (kafka.server)

```

图 4-43

makeLeader() 方法的第二个参数 PartitionState 包含了该方法使用到的数，其定义如下：

```

public class PartitionState {
    public final int controllerEpoch;
    public final int leader; // Leader 副本的 id, 实际上是 Leader 所在的 BrokerId
    public final int leaderEpoch;
    // ISR 集合，其中保存的是 ISR 集合中副本所在的 BrokerId
    public final List<Integer> isr;
    public final int zkVersion;
    // AR 集合，其中保存的是 AR 集合中副本所在的 BrokerId
    public final Set<Integer> replicas;
    ... ..
}

```

makeLeader() 方法会按照的 PartitionState 指定的信息，将 leader 字段指定的副本转换成 Leader 副本。makeLeader() 方法的返回值是 Boolean 类型，表示 Leader 副本是否发生了迁移，即新 Leader 副本与旧 Leader 副本在不同的 Broker 上，其代码如下：

```

def makeLeader(controllerId: Int, partitionStateInfo: PartitionState,
correlationId: Int): Boolean = {
    val (leaderHWIncremented, isNewLeader) = inWriteLock(leaderIsrUpdateLock) { // 加锁
        // 获取需要分配的 AR 集合
        val allReplicas = partitionStateInfo.replicas.asScala.map(_.toInt)
        // 记录 controllerEpoch
        controllerEpoch = partitionStateInfo.controllerEpoch
        // 步骤 1: 创建 AR 集合中所有副本对应的 Replica 对象
        allReplicas.foreach(replica => getOrCreateReplica(replica))
        // 步骤 2: 获取 ISR 集合
        val newInSyncReplicas = partitionStateInfo.isr.asScala.map(
            r => getOrCreateReplica(r)).toSet
    }
}

```

```

// 步骤3: 根据 allReplicas 更新 assignedReplicas 集合
(assignedReplicas().map(_._brokerId) -- allReplicas).foreach(removeReplica(_))
// 步骤4: 更新 Partition 字段
inSyncReplicas = newInSyncReplicas // 更新 ISR 集合
leaderEpoch = partitionStateInfo.leaderEpoch // 更新 leaderEpoch
zkVersion = partitionStateInfo.zkVersion // 更新 zkVersion

// 步骤5: 检测 Leader 是否发生变化
val isNewLeader =
    if (leaderReplicaIdOpt.isDefined && leaderReplicaIdOpt.get ==
localBrokerId) {
        false // Leader 所在的 Broker 并没有发生变化, 则为 false
    } else {
// Leader 之前并不在此 Broker 上, 即 Leader 发生变化, 更新 leaderReplicaIdOpt, 则为 true
        leaderReplicaIdOpt = Some(localBrokerId)
        true
    }
val leaderReplica = getReplica().get // 获取 Local Replica
if (isNewLeader) {
    // 步骤6: 初始化 Leader 的 highWatermarkMetadata
    // 如果 Leader 副本发生了迁移, 则表示 Leader 副本通过上面的步骤刚刚分配到此
    // Broker 上。可能是刚启动, 也可能是 Follower 副本成为 Leader 副本
    // 通过前面对 Replica 的分析, 我们知道 leaderReplica.highWatermarkMetadata
    // 字段是一个 LogOffsetMetadata 对象, 此时它只有 messageOffset 字段有值,
    // segmentBaseOffset 和 relativePositionInSegment 字段都是 -1, 需要初始化
    // 这两个字段值
    // convertHWToLocalOffsetMetadata() 方法底层是通过 Log.read() 方法实现的, 如
    // 果初始化失败, 则将 LogOffsetMetadata.messageOffset 重置为 -1, 另外两个字段
    // 的值重置为 0
    leaderReplica.convertHWToLocalOffsetMetadata()
    // 步骤7: 重置所有 Remote Replica 的 LEO 值为 -1
    assignedReplicas.filter(_._brokerId != localBrokerId)
        .foreach(_._updateLogReadResult(LogReadResult.UnknownLogReadResult))
}
// 步骤8: 尝试更新 HW
(maybeIncrementLeaderHW(leaderReplica), isNewLeader)
}
// 如果 Leader 副本的 HW 增加了, 则可能有 DelayedFetch 满足执行条件, 所以这里调用

```

```

// tryCompleteDelayedRequests() 方法会检查 delayedProducePurgatory、
// delayedFetchPurgatory 中 key 为当前分区的延时操作
if (leaderHWIncremented) {
    tryCompleteDelayedRequests()
}
isNewLeader
}

```

Partition.maybeIncrementLeaderHW() 方法会尝试后移 Leader 副本的 HW。当 ISR 集合发生增减或是 ISR 集合中任一副本的 LEO 发生变化时，都可能导致 ISR 集合中最小的 LEO 变大，所以这些情况都要调用 maybeIncrementLeaderHW() 方法进行检测，如图 4-44 所示。

▼ **Partition.maybeIncrementLeaderHW(Replica) (kafka.cluster)**

- ▶ ① Partition.maybeExpandIsr(int) (kafka.cluster)
- ▶ ① Partition.makeLeader(int, PartitionState, int) (kafka.cluster)
- ▶ ① Partition.maybeShrinkIsr(long) (kafka.cluster)
- ▶ ① Partition.appendMessagesToLeader(ByteBufferMessageSet, int) (kafka.cluster)

图 4-44

```

private def maybeIncrementLeaderHW(leaderReplica: Replica): Boolean = {
    // 获取 ISR 集合中所有副本的 LEO
    val allLogEndOffsets = inSyncReplicas.map(_.logEndOffset)
    // 将 ISR 集合中最小的 LEO 作为新 HW
    val newHighWatermark = allLogEndOffsets.min(new LogOffsetMetadata.
OffsetOrdering)
    val oldHighWatermark = leaderReplica.highWatermark // 获取当前 HW
    // 比较新旧两个 HW 的值，决定是否更新 HW
    if (oldHighWatermark.messageOffset < newHighWatermark.messageOffset
        || oldHighWatermark.onOlderSegment(newHighWatermark)) {
        leaderReplica.highWatermark = newHighWatermark // 更新 HW
        true
    } else {
        false
    }
}

```

Partition.makeFollower() 方法与 Partition.makeLeader() 方法类似，也是处理

LeaderAndIsrRequest 的环节之一。它的功能是按照 PartitionState 指定的信息，将 Local Replica 设置为 Follower 副本，其调用栈如图 4-45 所示。

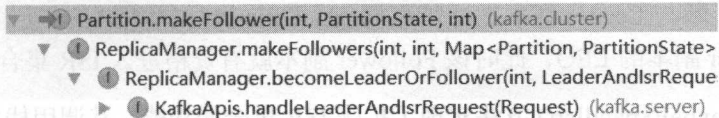


图 4-45

```

def makeFollower(controllerId: Int, partitionStateInfo: PartitionState,
correlationId: Int): Boolean = {
  inWriteLock(leaderIsrUpdateLock) { // 加锁
    val allReplicas = partitionStateInfo.replicas.asScala.map(_.toInt)
    val newLeaderBrokerId: Int = partitionStateInfo.leader
    controllerEpoch = partitionStateInfo.controllerEpoch

    // 创建对应的 Replica 对象
    allReplicas.foreach(r => getOrCreateReplica(r))
    // 与 makeLeader() 一样，根据 partitionStateInfo 信息更新 AR 集合
    (assignedReplicas().map(_.brokerId) -- allReplicas).foreach(removeReplica(_))

    // 空集合，ISR 集合在 Leader 副本上进行维护，Follower 副本上不维护 ISR 集合信息
    inSyncReplicas = Set.empty[Replica]
    leaderEpoch = partitionStateInfo.leaderEpoch // 更新 leaderEpoch 字段
    zkVersion = partitionStateInfo.zkVersion // 更新 zkVersion 字段
    // 检测 Leader 是否发生变化
    if (leaderReplicaIdOpt.isDefined && leaderReplicaIdOpt.get == newLeaderBrokerId)
    {
      false
    } else {
      // 更新 leaderReplicaIdOpt 字段
      leaderReplicaIdOpt = Some(newLeaderBrokerId)
      true
    }
  }
}

```


ISR 集合管理

Partition 除了对副本的 Leader/Follower 角色进行管理, 还需要管理 ISR 集合。随着 Follower 副本不断与 Leader 副本进行消息同步, Follower 副本的 LEO 会逐渐后移, 并最终追赶上 Leader 副本的 LEO, 此时该 Follower 副本就有资格进入 ISR 集合。

Partition.maybeExpandIsr() 方法实现了扩张 ISR 集合的功能, 其调用栈如图 4-46 所示, 它是在 updateFollowerLogReadResults() 方法中被调用的, 在前面介绍 DelayedFetch 的处理流程时提到过此方法的功能, 该方法用于处理来自 Follower 的 FetchRequest。

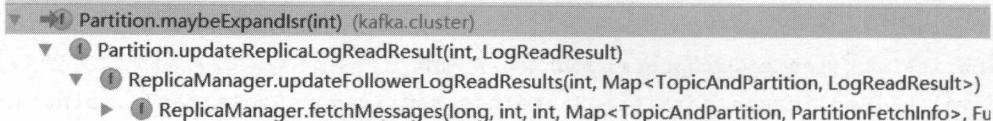


图 4-46

maybeExpandIsr() 方法的代码如下所示。

```

def maybeExpandIsr(replicaId: Int) {
  val leaderHWIncremented = inWriteLock(leaderIsrUpdateLock) { // 加锁
    // 只有 Leader 副本才会管理 ISR, 所以先获取 Leader 副本对应的 Replica 对象
    leaderReplicaIfLocal() match {
      case Some(leaderReplica) =>
        val replica = getReplica(replicaId).get
        val leaderHW = leaderReplica.highWatermark // 获取当前 HW

        // 下面检测的三个条件依次是: Follower 副本不在 ISR 集合中, AR 集合中可以查找到
        // Follower 副本, Follower 副本的 LEO 已经追赶上 HW
        if (!inSyncReplicas.contains(replica) &&
            assignedReplicas.map(_.brokerId).contains(replicaId) &&
            replica.logEndOffset.offsetDiff(leaderHW) >= 0) {
          // 将该 Follower 副本添加到 ISR 集合, 形成新的 ISR 集合
          val newInSyncReplicas = inSyncReplicas + replica
          // 将新 ISR 集合的信息上传到 ZooKeeper 中保存
          // 并更新 Partition.inSyncReplicas 字段
          updateIsr(newInSyncReplicas)
        }
        maybeIncrementLeaderHW(leaderReplica) // 尝试更新 HW
      case None => false
    }
  }
}
  
```

```

    }
  }
  if (leaderHWIncremented) // 尝试执行延时任务
    tryCompleteDelayedRequests()
}

```

在分布式系统中，各个节点通过网络交互可能出现阻塞和延迟，导致 ISR 集合中的部分 Follower 副本无法与 Leader 副本进行同步。如果此时 `ProducerRequest` 的 `acks` 字段设置为 -1，则会长时间等待。为了避免出现这种情况，Partition 会对 ISR 集合进行缩减，此功能在 `Partition.maybeShrinkIsr()` 方法中实现。在 `ReplicaManager` 中使用定时任务周期性地调用 `maybeShrinkIsr()` 方法检查 ISR 集合中 Follower 副本与 Leader 副本之间的同步差距，并对 ISR 集合进行缩减。

```

def maybeShrinkIsr(replicaMaxLagTimeMs: Long) {
  val leaderHWIncremented = inWriteLock(leaderIsrUpdateLock) { // 加锁
    // 调用 leaderReplicaIfLocal() 方法获取 Leader 副本对应的 Replica 对象
    leaderReplicaIfLocal() match {
      case Some(leaderReplica) =>
        // 通过检测 Follower 副本的 lastCaughtUpTimeMsUnderlying 字段，找出已滞后的
        // Follower 副本集合，该滞后集合中的 Follower 副本会被剔除了 ISR 集合
        // 无论是长时间没有与 Leader 副本进行同步还是其 LEO 与 HW 相差太大，都可以从此字
        // 段反映出来
        val outOfSyncReplicas = getOutOfSyncReplicas(leaderReplica,
            replicaMaxLagTimeMs)
        if (outOfSyncReplicas.size > 0) {
          // 将 outOfSyncReplicas 从 ISR 集合中删除，生成新的 ISR 集合
          val newInSyncReplicas = inSyncReplicas -- outOfSyncReplicas
          // 将新 ISR 集合的信息上传到 ZooKeeper 中保存，并更新 inSyncReplicas 字段
          updateIsr(newInSyncReplicas)
          maybeIncrementLeaderHW(leaderReplica) // 更新 Leader 的 HW
        } else {
          false
        }
      case None => false
    }
  }
  if (leaderHWIncremented)
    tryCompleteDelayedRequests() // 尝试执行延时任务
}

```

有一点需要读者注意，在 ISR 集合发生增减的时候，都会将最新的 ISR 集合保存在 ZooKeeper 中，具体的保存路径是：/brokers/topics/[topic_name]/partitions/[partitionId]/state。后面介绍的 KafkaController 会监听此路径中数据的变化。

追加消息

在分区中，只有 Leader 副本能够处理读写请求。Partition.appendMessagesToLeader() 方法提供了向 Leader 副本对应的 Log 中追加消息的功能。在前面介绍的 DelayedProduce 处理流程中，ReplicaManager.appendToLocalLog() 方法就是基于此方法实现的，调用关系如图 4-47 所示。

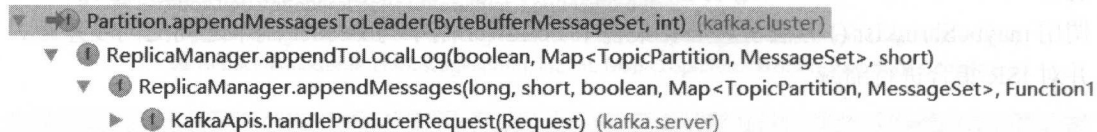


图 4-47

Partition.appendMessagesToLeader() 方法的代码如下：

```

def appendMessagesToLeader(messages: ByteBufferMessageSet, requiredAcks:
Int = 0) = {
  val (info, leaderHWIncremented) = inReadLock(leaderIsrUpdateLock) {
    // 获取 Leader 副本对应的 Replica 对象
    val leaderReplicaOpt = leaderReplicaIfLocal()
    leaderReplicaOpt match {
      case Some(leaderReplica) =>
        val log = leaderReplica.log.get
        // 获取配置指定的最小 ISR 集合大小的限制
        val minIsr = log.config.minInSyncReplicas
        val inSyncSize = inSyncReplicas.size

        // 如果当前 ISR 集合中副本的数量小于配置的最小限制，且生产者要求较高的可用性，
        // 则不能追加消息，生产者将收到一个 NotEnoughReplicasException 异常
        if (inSyncSize < minIsr && requiredAcks == -1) {
          // 抛出异常（略）
        }

        // 写入 Leader 副本对应的 Log
        val info = log.append(messages, assignOffsets = true)
    }
  }
}
  
```



```

        // 尝试执行对应的 DelayedFetch
        replicaManager.tryCompleteDelayedFetch(new
            TopicPartitionOperationKey(this.topic, this.partitionId))
        // 尝试增加 Leader 的 HW
        (info, maybeIncrementLeaderHW(leaderReplica))
    case None => ... ..// 抛出异常 (略)
}
}
if (leaderHWIncremented)
    tryCompleteDelayedRequests() // 尝试执行延时任务
info
}

```

checkEnoughReplicasReachOffset

在检测 DelayedProduce 的执行条件时，简单提到了 Partition.checkEnoughReplicasReachOffset() 方法，此方法会检测其参数指定的消息是否已经被 ISR 集合中所有 Follower 副本同步。

```

def checkEnoughReplicasReachOffset(requiredOffset: Long): (Boolean, Short)
= {
    leaderReplicaIfLocal() match { // 获取 Leader 副本对应的 Replica
    case Some(leaderReplica) =>
        val curInSyncReplicas = inSyncReplicas // 获取当前 ISR 集合
        val minIsr = leaderReplica.log.get.config.minInSyncReplicas
        // 比较 HW 与消息的 offset
        if (leaderReplica.highWatermark.messageOffset >= requiredOffset) {
            if (minIsr <= curInSyncReplicas.size) { // 检测 ISR 集合大小是否合法
                (true, Errors.NONE.code)
            } else {
                // ISR 集合太小，返回相应错误码
                (true, Errors.NOT_ENOUGH_REPLICAS_AFTER_APPEND.code)
            }
        } else
            (false, Errors.NONE.code)
    case None =>
        (false, Errors.NOT_LEADER_FOR_PARTITION.code)
    }
}
}

```

Broker 将其上的那些分区的副本切换成 Leader 角色，哪些分区的副本切换成 Follower 角色。

在 Partition 中还提供了 delete() 方法用来删除对应分区在此 Broker 上的 Log 文件，同时也会清空其 ISR、AR 等集合，代码比较简单，请读者参考源码学习。

4.5.3 ReplicaManager

分析完 Replica 和 Partition 的相关代码，本节介绍 ReplicaManager 组件的相关功能。在一个 Broker 上可能分布着多个 Partition 的副本信息，ReplicaManager 的主要功能是管理一个 Broker 范围内的 Partition 信息。ReplicaManager 的实现依赖于前面介绍的日志存储子系统、DelayedOperationPurgatory、KafkaScheduler 等组件，底层依赖于 Partition 和 Replica，如图 4-48 所示。

ReplicaManager 中各个字段的含义和功能如下所述。

- logManager: LogManager 对象，对分区的读写操作都委托给底层的日志存储子系统。
- scheduler: KafkaSchedule 对象，用于执行 ReplicaManager 中的周期性定时任务。在 ReplicaManager 中总共有三个周期性任务，它们分别是 highwatermark-checkpoint 任务、isr-expiration 任务、isr-change-propagation 任务。
- controllerEpoch: 记录 KafkaController 的年代信息，当重新选举 Controller Leader 时该字段值会递增。之后，在 ReplicaManager 处理来自 KafkaController 的请求时，会先检测请求中携带的年代信息是否等于 controllerEpoch 字段的值，这就避免接收旧 Controller Leader 发送的请求。这种设计方式在分布式系统中比较常见。
- localBrokerId: 当前 Broker 的 id，主要用于查找 Local Replica。
- allPartitions: Pool[(String, Int), Partition] 类型，其中保存了当前 Broker 上分配的所有 Partition 信息。这里需要注意 Pool 的 valueFactory，当从 Pool 查找不到指定 key 时，则使用 valueFactory 创建一个默认 value 值放入 Pool 并返回。如果读者了解 ThreadLocal 的 initialValue() 方法或 ThreadPoolExecutor 中 ThreadFactory 的相关设计，可以类比理解。

```
private val allPartitions = new Pool[(String, Int), Partition](
    valueFactory = Some { case (t, p) =>
        new Partition(t, p, time, this) // 创建 Partition 对象
    })
```

- replicaFetcherManager: 在 ReplicaFetcherManager 中管理了多个 ReplicaFetcherThread 线程，ReplicaFetcherThread 线程会向 Leader 副本发送 FetchRequest 请求来获取消息，

实现 Follower 副本与 Leader 副本同步。ReplicaFetcherManager 对象在 ReplicaManager 初始化时被创建，后面会详细介绍 ReplicaFetcherManager 与 ReplicaFetcherThread 的功能。

```
val replicaFetcherManager = new ReplicaFetcherManager(...)
```

- **highWatermarkCheckpoints**: Map[String, OffsetCheckpoint] 类型，用于缓存每个 log 目录与 OffsetCheckpoint 之间的对应关系，OffsetCheckpoint 记录了对应 log 目录下的 replication-offset-checkpoint 文件，该文件中记录了 data 目录下每个 Partition 的 HW。ReplicaManager 中的 highwatermark-checkpoint 任务会定时更新 replication-offset-checkpoint 文件的内容。

```
val highWatermarkCheckpoints = config.logDirs.map(dir => {
  new File(dir).getAbsolutePath, new OffsetCheckpoint(new File(dir,
    ReplicaManager.HighWatermarkFilename)))}.toMap
```

- **isrChangeSet**: Set[TopicAndPartition] 类型，用于记录 ISR 集合发生变化的分区信息。
- **delayedProducePurgatory**、**delayedFetchPurgatory**: 用于管理 DelayedProduce 和 DelayedFetch 的 DelayedOperationPurgatory 对象。
- **zkUtils**: 操作 ZooKeeper 的辅助类。

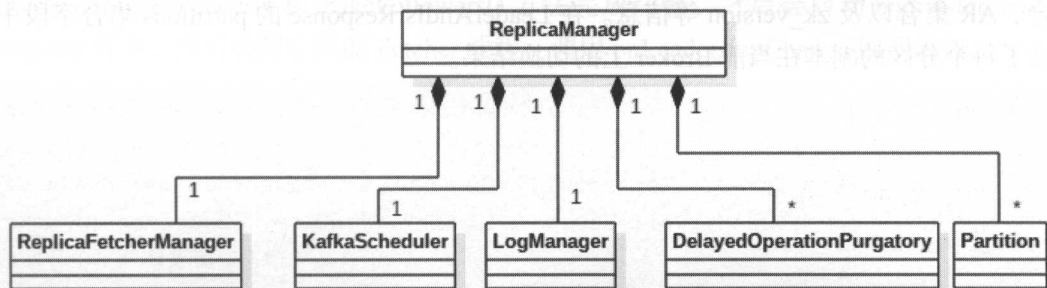


图 4-48

副本角色切换

在 Kafka 集群中会选举一个 Broker 成为 KafkaController 的 Leader，它负责管理整个 Kafka 集群。Controller Leader 根据 Partition 的 Leader 副本和 Follower 副本的状态向对应的 Broker 节点发送 LeaderAndIsrRequest，这个请求主要用于副本的角色切换，即指导 Broker 将其上的哪些分区的副本切换成 Leader 角色，哪些分区的副本切换成 Follower 介绍。

LeaderAndIsrRequest 首先由 `KafkaApis.handleLeaderAndIsrRequest()` 方法进行处理，其核心逻辑是通过 `ReplicaManager` 提供的 `becomeLeaderOrFollower()` 方法实现的，而 `becomeLeaderOrFollower()` 又依赖于上一小节介绍的 `Partition.makeLeader()` 方法和 `makeFollower()` 方法，上述调用关系如图 4-49 所示。

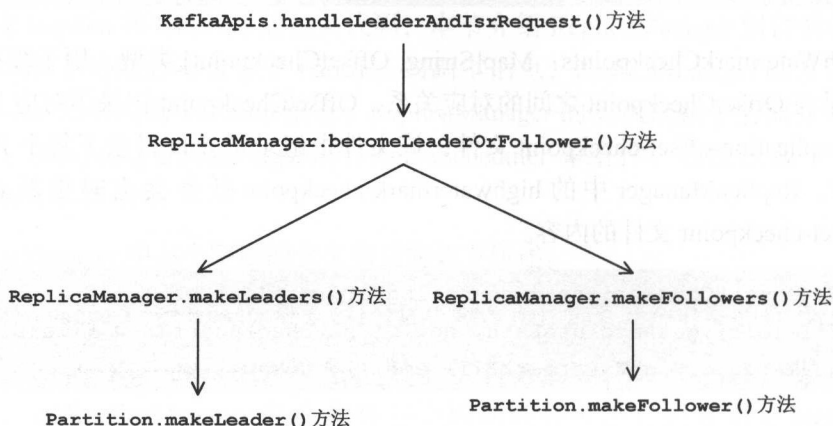


图 4-49

在开始分析 `becomeLeaderOrFollower()` 方法前，先来介绍一下 `LeaderAndIsrRequest` 和 `LeaderAndIsrResponse` 的格式，如图 4-50 所示。在 `LeaderAndIsrRequest` 中比较重要的是 `partition_states` 集合这个字段，其中包含了每个分区的 Leader 副本所在的 `BrokerId`、ISR 集合、AR 集合以及 `zk_version` 等信息。在 `LeaderAndIsrResponse` 的 `partitions` 集合字段中记录了每个分区的副本在当前 `Broker` 上的切换结果。

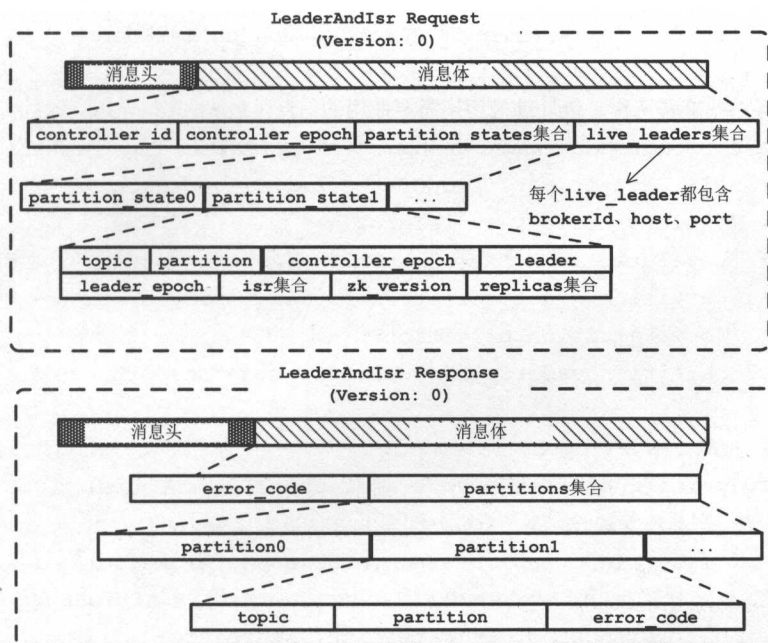


图 4-50

`ReplicaManager.becomeLeaderOrFollower()` 方法的主要逻辑是：获取（或创建）指定的 `Partition` 对象，根据 `partitionStates` 的信息对其切换成 `Leader/Follower` 的副本进行分类，并分别调用 `makeLeader()` 和 `makeFollowers()` 方法完成切换。之后会启动 `highwatermark-checkpoint` 任务，然后关闭空闲的 `Fetcher` 线程，调用 `onLeadershipChange` 回调函数。

```
def becomeLeaderOrFollower(correlationId: Int,
    leaderAndISRRequest: LeaderAndIsrRequest,
    metadataCache: MetadataCache, onLeadershipChange: (Iterable[Partition],
    Iterable[Partition]) => Unit): BecomeLeaderOrFollowerResult = {
    ... ..// 日志操作（略）
    replicaStateChangeLock synchronized { // 加锁
        // 统计返回的错误码
        val responseMap = new mutable.HashMap[TopicPartition, Short]
        // 检查 controllerEpoch
        if (leaderAndISRRequest.controllerEpoch < controllerEpoch) {
            ... ..// 日志操作（略），并直接返回 STALE_CONTROLLER_EPOCH 错误码
        } else {
            val controllerId = leaderAndISRRequest.controllerId
```



```

controllerEpoch = leaderAndISRRequest.controllerEpoch

// 进行一些准备工作, 统计进行切换需要使用的信息 (PartitionState)
val partitionState = new mutable.HashMap[Partition, PartitionState]()
leaderAndISRRequest.partitionStates.asScala.foreach {
  case (topicPartition, stateInfo) =>
    // 从 allPartitions 中获取 Partition 对象, 找不到就创建新 Partition 对象
    val partition = getOrCreatePartition(topicPartition.topic,
      topicPartition.partition)
    val partitionLeaderEpoch = partition.getLeaderEpoch()

    // 检测 leaderEpoch
    if (partitionLeaderEpoch < stateInfo.leaderEpoch) {
      // 判断该分区的副本是否被分配到了当前的 Broker
      if (stateInfo.replicas.contains(config.brokerId))
        // 保留与当前 Broker 相关的 Partition 以及 PartitionState
        partitionState.put(partition, stateInfo)
      else {
        ... ..// 日志操作 (略), 此分区对应错误码是 UNKNOWN_TOPIC_OR_PARTITION
      }
    } else {
      ... ..// 日志操作 (略), 此分区对应错误码是 STALE_CONTROLLER_EPOCH
    }
}

// 根据 PartitionState 中指定的角色进行分类
val partitionsToBeLeader = partitionState.filter {
  case (partition, stateInfo) => stateInfo.leader == config.brokerId
}

val partitionsToBeFollower = (partitionState -- partitionsToBeLeader.
keys)

// 将指定分区的副本切换成为 Leader 副本
val partitionsBecomeLeader = if (!partitionsToBeLeader.isEmpty) {
  makeLeaders(controllerId, controllerEpoch, partitionsToBeLeader,
    correlationId, responseMap)
}

// 将指定分区的副本切换成为 Follower 副本

```

```

val partitionsBecomeFollower = if (!partitionsToBeFollower.isEmpty) {

    makeFollowers(controllerId, controllerEpoch, partitionsToBeFollower,
        correlationId, responseMap, metadataCache)

}

// 启动 highwatermark-checkpoint 任务, 后面详述此任务的实现
if (!hwThreadInitialized) {
    startHighWaterMarksCheckPointThread()
    hwThreadInitialized = true
}

// 关闭 PeplcaFetcherManageridle 中空闲的 Fetcher 线程, 后面详述
replicaFetcherManager.shutdownIdleFetcherThreads()
// 回调函数, 后面详述
onLeadershipChange(partitionsBecomeLeader, partitionsBecomeFollower)
BecomeLeaderOrFollowerResult(responseMap, Errors.NONE.code)
}
}
}

```

ReplicaManager.makeLeaders() 方法会将指定分区的 Local Replica 切换为 Leader 的。如果是从 Follower 副本切换成 Leader 副本, 那么要先停止相关的 Fetcher 线程, 之后调用 Partition.makeLeader() 方法完成切换。

```

private def makeLeaders(controllerId: Int, epoch: Int,
    partitionState: Map[Partition, PartitionState], correlationId: Int,
    responseMap: mutable.Map[TopicPartition, Short]): Set[Partition] = {
    ... .. // 初始化每个分区对应的错误码为 Errors.NONE (略)
    val partitionsToMakeLeaders: mutable.Set[Partition] = mutable.Set()
    try {
        // 在此 Broker 上的副本之前可能是 Follower, 所以要先暂停对这些副本的 fetch 操作
        replicaFetcherManager.removeFetcherForPartitions(
            partitionState.keySet.map(new TopicAndPartition(_)))

        partitionState.foreach {
            case (partition, partitionStateInfo) =>
                // 调用 Partition.makeLeader() 方法, 将分区的 Local Replica 切换为 Leader 副本

```

```

        if (partition.makeLeader(controllerId, partitionStateInfo,
correlationId))
            // 记录成功从其他状态（第一次启动或 Follower 副本）切换成 Leader 副本的分区
            partitionsToMakeLeaders += partition
        else // 日志操作（略）
        }
    } catch { // 异常处理（略） }
    partitionsToMakeLeaders
}

```

`ReplicaManager.makeFollowers()` 方法会将分区的 Local Replica 切换为 Follower 副本。如果是从 Leader 副本切换为 Follower 副本，需要先检测新 Leader 副本是否存活，然后决定是否进行切换。切换结束后，需要先停止与旧 Leader 副本同步的 `Fetcher` 线程，然后对 Log 进行相应的截断处理，再启动与新 Leader 副本的 `Fetcher` 线程进行同步。最后还会尝试完成分区相关的 `DelayedOperation`。

```

private def makeFollowers(controllerId: Int, epoch: Int,
                        partitionState: Map[Partition, PartitionState],
correlationId: Int,
                        responseMap: mutable.Map[TopicPartition, Short],
                        metadataCache: MetadataCache): Set[Partition] = {
    .....// 初始化每个分区对应的错误码为 Errors.NONE（略）
    val partitionsToMakeFollower: mutable.Set[Partition] = mutable.Set()
    try {
        partitionState.foreach {
            case (partition, partitionStateInfo) =>
                // 检测新 Leader 所在的 Broker 是否存活
                val newLeaderBrokerId = partitionStateInfo.leader
                metadataCache.getAliveBrokers.find(_.id == newLeaderBrokerId) match
                {
                    case Some(leaderBroker) =>
                        // 调用 Partition.makeFollower() 方法，将分区的 Local Replica 切换为 Follower 副本
                        if (partition.makeFollower(controllerId, partitionStateInfo,
correlationId))
                            // 记录成功从其他状态（第一次启动或 Leader）切换到 Follower 副本的分区
                            partitionsToMakeFollower += partition
                        else

```



```

... .. // 日志操作 (略)
case None =>
    // 即使是 Leader 副本所在 Broker 不可用, 也要创建 Local Replica, 主要是
    // 为了在 checkpoint 文件中记录此分区的 HW, 感兴趣的读者可以参考
    // KAFKA-1647 这个 Bug 的介绍和处理
    partition.getOrCreateReplica()
}
}

// 停止与旧 Leader 同步的 fetch 线程
replicaFetcherManager.removeFetcherForPartitions(
    partitionsToMakeFollower.map(new TopicAndPartition(_)))

// 由于 Leader 副本已发生变化, 所以新旧 Leader 副本在 HW~LEO 之间的消息可能是不一致
// 的, 但 HW 之前的消息是一致的, 所以将 Log 截断到 HW. 对于可能出现的 “Unclean
// leader election” 场景, 在后面会介绍如何处理
logManager.truncateTo(partitionsToMakeFollower.map(
    partition => (new TopicAndPartition(partition),
        partition.getOrCreateReplica().highWatermark.messageOffset))).
toMap()

// 尝试完成该分区相关的 DelayedOperation
partitionsToMakeFollower.foreach { partition =>
    val topicPartitionOperationKey = new TopicPartitionOperationKey(part
ition.topic,
        partition.partitionId)
    tryCompleteDelayedProduce(topicPartitionOperationKey)
    tryCompleteDelayedFetch(topicPartitionOperationKey)
}

if (isShuttingDown.get()) { // 检测 ReplicaManager 的运行状态
    ... .. // 日志输出 (略)
} else {
    // 重新开启与新 Leader 副本同步的 Fetcher 线程
    val partitionsToMakeFollowerWithLeaderAndOffset =
    partitionsToMakeFollower.map(partition =>
        new TopicAndPartition(partition) -> BrokerAndInitialOffset(

```



```

        metadataCache.getAliveBrokers.find(_.id == partition.
leaderReplicaIdOpt.get)
            .get.getBrokerEndPoint(config.interBrokerSecurityProtocol),
            partition.getReplica().get.logEndOffset.messageOffset)
        ).toMap

        replicaFetcherManager.addFetcherForPartitions(
            partitionsToMakeFollowerWithLeaderAndOffset)
    }
} catch {
    // 异常处理(略)
}
partitionsToMakeFollower
}

```

Partition.makeLeader() 和 makeFollower() 方法在前面已经分析过了。ReplicaFetcherManager 类与 MetadataCache 类的相关内容会在后面详细分析。到此为止，分区 Leader/Follower 副本的切换流程和具体实现就分析完了。

追加 / 读取消息

当 Local Replica 切换为 Leader 副本之后，就可以处理生产者发送的 ProducerRequest，将消息写入到 Log 中。在前面分析 DelayedProduce 的处理流程时，简单介绍了 ReplicaManager.appendMessages() 方法，当时着重关注了与 DelayedProduce 相关的处理以及 sendResponseCallback 回调方法的实现。这里详细分析 appendToLocalLog() 方法的实现，它首先会检测消息要写入的 Topic 是否为 Kafka 的内部 Topic（目前 Kafka 只有 Offsets Topic 一个内部 Topic），如果是内部 Topic 则需要检测是否允许对内部 Topic 进行追加，最终调用 Partition.appendMessagesToLeader() 方法完成消息追加。appendToLocalLog() 方法的第二个参数记录了每个分区需要追加的消息集合。

```

private def appendToLocalLog(internalTopicsAllowed: Boolean,
messagesPerPartition: Map[TopicPartition, MessageSet],
requiredAcks: Short): Map[TopicPartition, LogAppendResult] = {
    messagesPerPartition.map { // 对消息进行迭代
        case (topicPartition, messages) => // 每次迭代得到一个分区以及对应的消息集合
        // 检测目标 Topic 是否是 Kafka 的内部 Topic，例如 “__consumer_offsets” Topic
        // 如果是内部 Topic，则根据 internalTopicsAllowed 决定是否可以向内部 Topic 写入消息
    }
}

```

```

if (Topic.isInternal(topicPartition.topic) && !internalTopicsAllowed) {
    (topicPartition, LogAppendResult(LogAppendInfo.UnknownLogAppendInfo,
        Some(new InvalidTopicException(...))))
} else {
    try {
        // 从 allPartitions 集合中获取对应的 Partition 对象
        val partitionOpt = getPartition(topicPartition.topic,
            topicPartition.partition)
        val info = partitionOpt match {
            case Some(partition) =>
                // 调用 Partition.appendMessagesToLeader() 方法，将消息写入对应的 Log 中
                partition.appendMessagesToLeader(
                    messages.asInstanceOf[ByteBufferMessageSet],
                    requiredAcks)
            case None => // 找不到 Partition 对象，抛出异常（略）
        }
        ... .. // 统计追加的消息数量（略）
        (topicPartition, LogAppendResult(info)) // 返回每个分区写入消息的结果
    } catch {
        // 异常处理（略）
    }
}
}
}
}

```

Leader 副本的另一个重要功能是处理 `FetchRequest` 进行消息读取。在分析 `DelayedFetch` 的处理流程时，简单介绍了 `ReplicaManager.fetchMessages()` 方法，当时着重关注了 `DelayedFetch` 相关的处理以及 `sendResponseCallback` 回调方法的实现，这里来分析 `readFromLocalLog()` 方法以及 `updateFollowerLogReadResults()` 的实现。`readFromLocalLog()` 方法的第一个参数表示是否只读取 Leader 副本的消息，只有处理来自 Debug 模式下的消费者的请求该参数才会为 `false`，正常运行模式下始终为 `true`；第二个参数表示是否只读取完成提交的消息（即 HW 之前的消息），如果是处理来自消费者的请求此参数为 `true`，Follower 副本对应 `false`；第三个参数记录了每个分区读取的起始 offset 位置和最大字节数。`readFromLocalLog()` 方法的代码如下：

```

def readFromLocalLog(fetchOnlyFromLeader: Boolean, readOnlyCommitted:
Boolean,
    readPartitionInfo: Map[TopicAndPartition, PartitionFetchInfo]):
    Map[TopicAndPartition, LogReadResult] = {
    readPartitionInfo.map {
        case (TopicAndPartition(topic, partition), PartitionFetchInfo(offset,
fetchSize))
=> val partitionDataAndOffsetInfo =
        try {
            // 获取要读取消息的副本, 根据 fetchOnlyFromLeader 来判断是否必须为 Leader 副本
            val localReplica = if (fetchOnlyFromLeader)
                getLeaderReplicaIfLocal(topic, partition)
            else getReplicaOrException(topic, partition)

            // 确定读取消息的 offset 上限, 根据 readOnlyCommitted 来判断是否为 HW
            val maxOffsetOpt = if (readOnlyCommitted)
                Some(localReplica.highWatermark.messageOffset)
            else
                None

            val initialLogEndOffset = localReplica.logEndOffset
            // logReadInfo 是 FetchDataInfo 类型对象, 其中包含 LogOffsetMetadata 和
            // 消息集 messageSet。LogOffsetMetadata 前面介绍过, 不再赘述
            val logReadInfo = localReplica.log match {
                // 从 Log 中读取消息
                case Some(log) => log.read(offset, fetchSize, maxOffsetOpt)
                case None => ... .. // 异常处理 (略)
            }

            // 是否已经读到 Log 的最后一条消息
            val readToEndOfLog = initialLogEndOffset.messageOffset -
                logReadInfo.fetchOffsetMetadata.messageOffset <= 0

            // 封装成 LogReadResult 对象返回
            LogReadResult(logReadInfo, localReplica.highWatermark.messageOffset,
                fetchSize, readToEndOfLog, None)
        } catch {
            // 异常处理 (略)

```



```

    }
    (TopicAndPartition(topic, partition), partitionDataAndOffsetInfo)
  }
}

```

当 ISR 集合中所有 Follower 副本都已经同步了某消息时，Kafka 认为消息已经成功提交，可以将 HW 后移。所以这里针对来自 Follower 副本的 `FetchRequest` 多了一步处理，即 `ReplicaManager.updateFollowerLogReadResults()` 方法，主要做下面 4 件事：

(1) 更新 Leader 副本上维护的 Follower 副本的各项状态，例如，更新 LEO、`lastCaughtUpTimeMsUnderlying` 等。

(2) 随着 Follower 副本不断 fetch 消息，最终追上 Leader 副本，可能对 ISR 集合进行扩张，如果 ISR 集合发生变化，则将新 ISR 集合的记录保存到 ZooKeeper 上。

(3) 检测是否需要后移 Leader 的 HW。

(4) 检测 `delayedProducePurgatory` 中相关 key 对应的 `DelayedProduce`，满足条件则将其执行完成。

`updateFollowerLogReadResults()` 方法的具体代码如下：

```

private def updateFollowerLogReadResults(replicaId: Int, readResults:
  Map[TopicAndPartition, LogReadResult]) {
  readResults.foreach { // 遍历上面介绍的读取 Log 的结果
    case (topicAndPartition, readResult) =>
      getPartition(topicAndPartition.topic, topicAndPartition.partition)
  match {
    case Some(partition) =>
      // 调用 Partition.updateReplicaLogReadResult() 方法，其中会更新
      // Follower 副本的状态调用 maybeExpandIsr() 方法尝试扩张 ISR 集合
      partition.updateReplicaLogReadResult(replicaId, readResult)

      // 尝试执行 DelayedProduce
      tryCompleteDelayedProduce(new TopicPartitionOperationKey(topicAndPartition))
    case None => .....// 输出警告日志 (略)
  }
}
}

```



```
// 下面是 Partition.updateReplicaLogReadResult() 方法的实现
def updateReplicaLogReadResult(replicaId: Int, logReadResult: LogReadResult) {
  getReplica(replicaId) match {
    case Some(replica) =>
      replica.updateLogReadResult(logReadResult) // 更新 Follower 副本的状态
      maybeExpandIsr(replicaId) // 检测 ISR 是否扩张, 并与 ZooKeeper 进行同步
    case None => ... ..// 抛出异常 (略)
  }
}
```

ReplicaManager 中与追加 / 读取消息相关的方法的实现介绍完了, 这些方法都是 Leader 副本对外提供的功能。下一节将介绍 Follower 副本如何发送 FetchRequest 与 Leader 进行同步。

消息同步

Follower 副本与 Leader 副本同步的功能由 ReplicaFetcherManager 组件实现。ReplicaFetcherManager 继承了 AbstractFetcherManager。AbstractFetcherManager 的继承和依赖关系如图 4-51 所示。

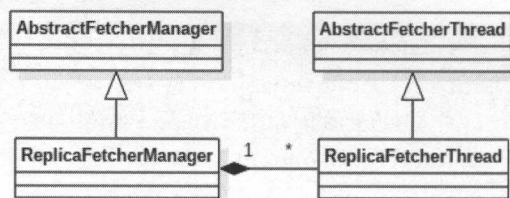


图 4-51

在 AbstractFetchManager 中使用 fetcherThreadMap 字段 (HashMap[BrokerAndFetcherId, AbstractFetcherThread] 类型) 管理 AbstractFetcherThread, 该 Map 的 key 值是 BrokerAndFetcherId 类型对象, 其中封装了 Broker 的网络位置信息 (brokerId、host、port 等) 以及对应的 Fetcher 线程的 id。AbstractFetcherManager 中还提供了 addFetcherForPartitions() 方法、removeFetcherForPartitions() 方法和 shutdownIdleFetcherThreads() 方法对 fetcherThreadMap 集合进行管理。

AbstractFetcherManager.addFetcherForPartitions() 方法会让 Follower 副本从指定的 offset 开始与 Leader 副本进行同步。该方法的参数涉及 BrokerAndInitialOffset 类, 它其中封装了 Broker 的网络位置信息以及同步的起始 offset。具体的同步逻辑交由 ReplicaFetcherThread

线程处理。

```
def addFetcherForPartitions(partitionAndOffsets: Map[TopicAndPartition,
    BrokerAndInitialOffset]) {
    mapLock synchronized {
        val partitionsPerFetcher = partitionAndOffsets.groupBy{
            case(topicAndPartition, brokerAndInitialOffset) =>
// 首先通过分区所属的 Topic 和分区编号计算得到对应的 Fetcher 线程 id, 然后与 Broker 的网
// 络位置信息组成 Key, 并进行分组。每组对应相同的 Fetcher 线程 (AbstractFetcherThread)
// 注意, 每个 Fetcher 线程只连接一个 Broker, 但可以为多个分区的 Follower 副本完成同步
            BrokerAndFetcherId(brokerAndInitialOffset.broker,
                getFetcherId(topicAndPartition.topic, topicAndPartition.
partition)))

// 按照 key 查找对应的 Fetcher 线程, 查找不到就创建新的 Fetcher 线程并启动
for ((brokerAndFetcherId, partitionAndOffsets) <- partitionsPerFetcher)
{
    var fetcherThread: AbstractFetcherThread = null
    fetcherThreadMap.get(brokerAndFetcherId) match {
        case Some(f) => fetcherThread = f // 查找到 Fetcher 线程
        case None => // 查找不到 Fetcher 线程的情况
            // createFetcherThread() 是抽象方法, 在子类 ReplicaFetcherManager 中实现
            fetcherThread = createFetcherThread(brokerAndFetcherId.
fetcherId,
                brokerAndFetcherId.broker)
            // 添加到 fetcherThreadMap 中管理并启动
            fetcherThreadMap.put(brokerAndFetcherId, fetcherThread)
            fetcherThread.start
        }

// 将分区信息以及同步起始位置传递给 Fetcher 线程, 并唤醒 Fetcher 线程, 开始同步
fetcherThreadMap(brokerAndFetcherId).addPartitions(
    partitionAndOffsets.map {
        case (topicAndPartition, brokerAndInitOffset) =>
            topicAndPartition -> brokerAndInitOffset.initOffset
    })
}
}
}
```

`AbstractFetcherManager.removeFetcherForPartitions()` 方法会停止指定 Follower 副本的同步操作，其代码如下：

```
def removeFetcherForPartitions(partitions: Set[TopicAndPartition]) {
  mapLock synchronized {
    for ((key, fetcher) <- fetcherThreadMap) {
      // 将 TopicAndPartition 信息从 Fetcher 线程中移除
      fetcher.removePartitions(partitions)
    }
  }
}
```

如果 Fetcher 线程不再为任何分区的 Follower 副本进行同步，则会在 `shutdownIdleFetcherThreads()` 方法中被停止，该方法会调用 Fetcher 线程的 `shutdown()` 方法。`ReplicaFetcherManager` 的代码比较简单，只提供了 `createFetcherThread()` 方法的实现，该实现会创建了一个 `ReplicaFetcherThread` 对象，逻辑比较简单，代码不再贴出来了。

了解了 `ReplicaManager` 如何管理 fetcher 线程，下面来分析 fetcher 线程发送 `FetchRequest`、处理 `FetchResponse` 的相关实现。`ReplicaFetcherThread` 是 fetcher 线程的具体实现，继承了 `AbstractFetcherThread` 抽象类。它的核心字段是 `partitionMap`（`HashMap[TopicAndPartition, PartitionFetchState]` 类型），维护了 `TopicAndPartition` 与 `PartitionFetchState` 之间的对应关系，`PartitionFetchState` 记录了对应分区的同步 offset 位置以及同步状态。

在 `AbstractFetcherThreadManager` 中调用了 `AbstractFetcherThread.addPartitions()` 方法和 `removePartitions()` 方法对 `partitionMap` 字段进行增删，同时会唤醒 Fetcher 线程进行同步。这两个方法的实现如下：

```
// 下面是 addPartitions() 方法的实现
def addPartitions(partitionAndOffsets: Map[TopicAndPartition, Long]) {
  partitionMapLock.lockInterruptibly() // 加锁
  try {
    for ((topicAndPartition, offset) <- partitionAndOffsets) {
      if (!partitionMap.contains(topicAndPartition)) // 检测指定分区是否已经存在
        partitionMap.put(
          topicAndPartition,
          if (PartitionTopicInfo.isOffsetInvalid(offset))
            new PartitionFetchState(handleOffsetOutOfRange(topicAndPartiti
on))
        )
    }
  }
}
```



```

        else new PartitionFetchState(offset)
    })
    partitionMapCond.signalAll() // 唤醒当前 fetcher 线程, 进行同步操作
} finally partitionMapLock.unlock()
}

// 下面是 removePartitions() 方法的实现
def removePartitions(topicAndPartitions: Set[TopicAndPartition]) {
    partitionMapLock.lockInterruptibly()
    try {
        topicAndPartitions.foreach { topicAndPartition =>
            partitionMap.remove(topicAndPartition)
            fetcherLagStats.unregister(topicAndPartition.topic, topicAndPartition.
partition)
        }
    } finally partitionMapLock.unlock()
}

```

`AbstractFetcherThread` 是 `ShutdownableThread` 线程的子类, 其核心业务代码在 `doWork()` 方法中, 其逻辑是: 根据 `partitionMap` 中维护的信息生成 `FetchRequest`, 如果当前没有需要进行同步的分区, 则会退避一段时间后重试; 如果有需要发送的 `FetchRequest`, 则执行 `processFetchRequest()` 方法进行处理。

```

override def doWork() {
    val fetchRequest = inLock(partitionMapLock) { // 加锁
        // 创建 FetchRequest 请求
        val fetchRequest = buildFetchRequest(partitionMap)
        if (fetchRequest.isEmpty) { // 没有 FetchRequest, 则退避一段时间后重试
            partitionMapCond.await(fetchBackOffMs, TimeUnit.MILLISECONDS)
        }
        fetchRequest
    }
    if (!fetchRequest.isEmpty)
        // 发送 FetchRequest 并处理 FetchResponse
        processFetchRequest(fetchRequest)
}

```

这里使用了模板方法模式, `buildFetchRequest()` 方法以及 `processFetchRequest()` 方法中

调用的很多方法都是由子类实现的抽象方法。在第3章已经介绍了 `FetchRequest` 的格式，Follower 副本与消费者的区别在于请求 `replica_id` 字段不再是“-1”而是 `replicaId`，这也是区分 `FetchRequest` 来源的标志。`ReplicaFetcherThread.buildFetchRequest()` 方法会在后面与 `handlePartitionsWithErrors()` 方法一起介绍。

`processFetchRequest()` 方法中定义了发送 `FetchRequest` 以及处理 `FetchResponse` 的步骤，但是这些步骤由子类 `ReplicaFetchRequest` 具体实现。

```
private def processFetchRequest(fetchRequest: REQ) {
  ... ..
  try {
    // 发送 FetchRequest 并等待 FetchResponse, fetch() 是抽象方法
    responseData = fetch(fetchRequest)
  } catch {
    ... ..// 异常处理, 线程退避一段时间(略)
  }
  if (responseData.nonEmpty) { // 处理 FetchResponse
    inLock(partitionMapLock) { // 加锁, 遍历每个 TopicAndPartition 对应的响应信息
      responseData.foreach { case (topicAndPartition, partitionData) =>
        val TopicAndPartition(topic, partitionId) = topicAndPartition
        partitionMap.get(topicAndPartition).foreach(currentPartitionFetchState =>
          // 从发送 FetchRequest 到收到 FetchResponse 这段同步时间内, offset 并未发生变化
          if (fetchRequest.offset(topicAndPartition) ==
              currentPartitionFetchState.offset) {
            Errors.forCode(partitionData.errorCode) match {
              case Errors.NONE =>
                try { // 获取返回的消息集合
                  val messages = partitionData.toByteBufferMessageSet
                  val validBytes = messages.validBytes // 验证
                  // 获取返回的最后一条消息的 offset
                  val newOffset = messages.shallowIterator.toSeq.
lastOption match {
                    case Some(m: MessageAndOffset) => m.nextOffset
                    case None => currentPartitionFetchState.offset
                  }
                  partitionMap.put(topicAndPartition,
                                // 更新 Fetch 状态
```

```

        new PartitionFetchState(newOffset))

        // 将从 Leader 副本获取的消息集合追加到 Log 中
        processPartitionData(topicAndPartition,
                               currentPartitionFetchState.offset,
partitionData)
    } catch {
        ... ..// 异常处理 (略)
    }
case Errors.OFFSET_OUT_OF_RANGE =>
// 若 Follower 副本请求的 offset 超出了 Leader 的 LEO, 则返回此错误码
try {
    // 生成新的 offset, handleOffsetOutOfRange() 是抽象方法
    val newOffset = handleOffsetOutOfRange(topicAndPartition)
    partitionMap.put(topicAndPartition,
                     // 更新 Fetch 状态
                     new PartitionFetchState(newOffset))
} catch {
    ... ..// 异常处理 (略)
}
case _ =>
// 返回其他错误码, 则进行收集后, 由 handlePartitionsWithErrors() 方法处理
    if (isRunning.get) {
        partitionsWithError += topicAndPartition
    }
}
}))
}
}
}
if (partitionsWithError.nonEmpty) {
    handlePartitionsWithErrors(partitionsWithError) // 抽象方法
}
}

```

现在我们跳转到 `ReplicaFetcherThread` 中, 分析上述各个抽象方法的具体实现。`ReplicaFetcherThread` 的核心字段是 `networkClient`, 在第2章已经对 `NetworkClient` 做了详细的介绍, 请读者回顾, 在这里它负责 Follower 副本与 Leader 副本之间的网络通信。但是,

ReplicaFetcherThread 通过 NetworkClientBlockingOps 这个辅助类对 NetworkClient 进行了封装，为 NetworkClient 提供了同步阻塞的使用方式。NetworkClientBlockingOps 辅助类提供了 blockingReady() 和 blockingSendAndReceive() 两个阻塞方法，它们的功能分别是阻塞等待直到指定 Node 处于 Ready 状态、发送请求后阻塞等待响应。

```
// 下面是 blockingReady() 方法的实现
def blockingReady(node: Node, timeout: Long)(implicit time: JTime):
Boolean = {
  client.ready(node, time.milliseconds()) || pollUntil(timeout) {
    (_, now) =>
    if (client.isReady(node, now))    // 检测 Node 是否 Ready
      true
    else if (client.connectionFailed(node))
      throw new IOException(s"Connection to $node failed") // 抛出异常
    else false
  }
}

// 下面是 blockingSendAndReceive() 方法的实现
def blockingSendAndReceive(request: ClientRequest)(implicit time: JTime):
ClientResponse = {
  client.send(request, time.milliseconds()) // 发送请求
  pollContinuously { responses =>
    // 找到上面的发送请求对应的响应
    val response = responses.find { response =>
      response.request.request.header.correlationId ==
        request.request.header.correlationId
    }
    ... ..// 连接断开的情况，抛出异常（略）
    response
  }
}
```

NetworkClientBlockingOps 中的 pollUntil() 方法和 pollContinuously() 都是通过递归调用 recursivePoll() 方法实现阻塞的，每次递归都会调用 NetworkClient.poll()，然后检测是否满足递归结束的条件。


```
// 下面是 pollUntil() 方法的实现
private def pollUntil(timeout: Long)(predicate: (Seq[ClientResponse],
Long) => Boolean)
    (implicit time: JTime): Boolean = {
    val methodStartTime = time.milliseconds()
    val timeoutExpiryTime = methodStartTime + timeout // 计算超时时间
    // recursivePoll() 方法的定义
    def recursivePoll(iterationStartTime: Long): Boolean = {
        val pollTimeout = timeoutExpiryTime - iterationStartTime
        val responses = client.poll(pollTimeout, iterationStartTime).asScala
        // 检测是否满足递归结束条件
        if (predicate(responses, iterationStartTime)) true // 连接建立
        else {
            val afterPollTime = time.milliseconds()
            // 未超时继续递归
            if (afterPollTime < timeoutExpiryTime) recursivePoll(afterPollTime)
            else false // 超时返回
        }
    }
    recursivePoll(methodStartTime) // 递归入口
}

// 下面是 pollContinuously() 方法的实现
private def pollContinuously[T](collect: Seq[ClientResponse] => Option[T])
    (implicit time: JTime): T = {
    def recursivePoll: T = {
        // 下面的 poll() 设置的超时时间虽然是 Long.MaxValue, 但是并不会永远阻塞,
        // 因为还有 ClientRequest 的超时时间
        val responses = client.poll(Long.MaxValue, time.milliseconds()).
asScala
        collect(responses) match {
            // 检测是否满足递归结束条件
            case Some(result) => result // 递归结束
            case None => recursivePoll // 继续递归
        }
    }
    recursivePoll // 递归入口
}
```


我们回到对 `ReplicaFetcherThread` 的分析。`ReplicaFetcherThread` 对 `fetch()` 方法完全依赖于 `NetworkClientBlockingOps.blockingReady()` 和 `blockingSendAndReceive()` 方法实现。它首先调用 `blockingReady()` 方法检测阻塞等待 Node 变成 Ready 状态，然后调用 `blockingSendAndReceive()` 方法发送 `FetchRequest` 并阻塞等待 `FetchResponse`。

```
protected def fetch(fetchRequest: FetchRequest): Map[TopicAndPartition,
PartitionData] = {
    val clientResponse = sendRequest(ApiKeys.FETCH, Some(fetchRequestVersion),
        fetchRequest.underlying)
    new FetchResponse(clientResponse.responseBody).responseData.asScala.map {
        case (key, value) =>
            TopicAndPartition(key.topic, key.partition) -> new PartitionData(value)
    }
}
// 下面是 ReplicaFetcherThread.sendRequest() 方法的实现
private def sendRequest(apiKey: ApiKeys, apiVersion: Option[Short],
    request: AbstractRequest): ClientResponse = {
    import kafka.utils.NetworkClientBlockingOps._
    val header = apiVersion.fold(networkClient.nextRequestHeader(apiKey))
        (networkClient.nextRequestHeader(apiKey, _))
    try {
        // 阻塞等待 Node 的状态变为 Ready, 超时则会抛出异常
        if (!networkClient.blockingReady(sourceNode, socketTimeout)(time))
            throw new SocketTimeoutException(s"Failed to connect within
        $socketTimeout ms")
        else {
            val send = new RequestSend(sourceBroker.id.toString, header,
            request.toStruct)
            val clientRequest = new ClientRequest(time.milliseconds(), true,
            send, null)
            // 发送请求并阻塞等待响应
            networkClient.blockingSendAndReceive(clientRequest)(time)
        }
    }
    catch {
        ... ..// 异常处理 (略)
    }
}
```

`ReplicaFetcherThread.processPartitionData()` 方法会将 `fetch()` 方法返回来的消息追加到 Follower 副本的 Log 中，并更新 Follower 副本的 HW，代码如下：

```
def processPartitionData(topicAndPartition: TopicAndPartition, fetchOffset:
Long,
    partitionData: PartitionData) {
  try {
    val TopicAndPartition(topic, partitionId) = topicAndPartition
    val replica = replicaMgr.getReplica(topic, partitionId).get
    val messageSet = partitionData.toByteBufferMessageSet
    ... ..// 边界检查（略）
    // 将消息追加到 Log 中，注意第二个参数，使用 Leader 已经为消息分配了 offset，
    // Follower 副本不再对消息分配 offset
    replica.log.get.append(messageSet, assignOffsets = false)
    val followerHighWatermark =
      replica.logEndOffset.messageOffset.min(partitionData.highWatermark)
    // 更新 Follower 副本的 HW
    replica.highWatermark = new LogOffsetMetadata(followerHighWatermark)
  } catch {
    // 异常处理（略）
  }
}
```

`handleOffsetOutOfRange()` 方法主要处理 Follower 副本请求的 offset 超出了 Leader 副本的 offset 范围，可能是超过了 Leader 的 LEO，也可能是小于 Leader 的最小 offset（`startOffset`）。当发生“Unclean leader election”时就可能出现第一种情况，这种场景简单来说就是将不在 ISR 集合中的 Follower 副本被选举成为了 Leader 副本，发生此场景的过程如下：

（1）一个 Follower 副本发生宕机，而 Leader 副本不断接收来自生产者的消息并追加到 Log 中，此时 Follower 副本因为宕机并没有与 Leader 副本进行同步。

（2）此 Follower 副本重新上线，在它与 Leader 完全同步之前，它没有资格进入 ISR 集合。假设 ISR 集合中的 Follower 副本在此时全部宕机，只能选举此 Follower 副本为新 Leader 副本。

（3）之后，旧 Leader 重新上线成为 Follower 副本，此时就会出现 Follower 副本的 LEO 超越了 Leader 副本的 LEO 值的场景。

handleOffsetOutOfRange() 方法针对“Unclean leader election”场景的处理如下：

```
// 发送 ListOffsetRequest, 获取 Leader 副本的 LEO, 使用的是阻塞的方式
val leaderEndOffset: Long = earliestOrLatestOffset(topicAndPartition,
    ListOffsetRequest.LATEST_TIMESTAMP, brokerConfig.brokerId)
// 判断是否 Leader 副本的 LEO 是否依然落后于 Follower 副本
if (leaderEndOffset < replica.logEndOffset.messageOffset) {
    // 首先, 根据配置决定是否需要停机
    if (!LogConfig.fromProps(brokerConfig.originals,
        AdminUtils.fetchEntityConfig(replicaMgr.zkUtils,
            ConfigType.Topic, topicAndPartition.topic)).uncleanLeaderElectionEnable)
    {
        System.exit(1)
    }
    ... .. // 输出警告信息 (略)
    // 将分区对应的 Log 截断到 Leader 副本的 LEO 的位置, 之后从此 offset 开始重新与 Leader
    // 进行同步
    replicaMgr.logManager.truncateTo(Map(topicAndPartition -> leaderEndOffset))
    leaderEndOffset // 返回下次获取消息的 offset
}
```

在 Follower 发送 ListOffsetRequest 期间, 新 Leader 可能不断追加消息, 新 Leader 的 LEO 落后于 Follower 的 LEO 的场景得到改变, 此时就不再需要进行截断操作了, Follower 可以继续从其 LEO 与 Leader 进行同步。有的读者可能会说, 这样新 Leader 与 Follower 的消息可能存在不一致的情况, 如图 4-52 所示。

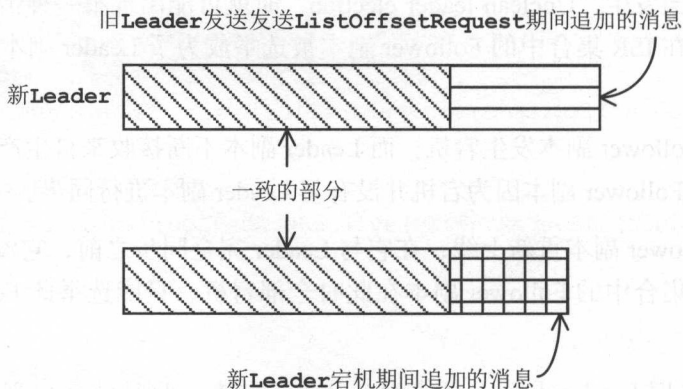


图 4-52

如果出现这种情况，我们就需要根据实际的业务逻辑进行权衡。如果业务逻辑可以忍受“Unclean leader election”场景带来的消息丢失和不一致，则可以将 `unclean.leader.election.enable` 配置为 `true`；如果业务逻辑不能接受这种场景，则关闭对“Unclean leader election”的支持，将停机作为优先选项。

如果 Follower 副本宕机后过了很长一段时间才重新上线，Leader 副本在此期间可能执行了多次 log retention 任务来删除陈旧的日志，这就可能导致 Leader 副本中第一条消息的 offset (`startOffset`) 大于 Follower 副本的 LEO。`handleOffsetOutOfRange()` 方法对于这种情况的处理代码如下：

```
// 发送 ListOffsetRequest, 获取 Leader 副本的 startOffset, 使用的是阻塞的方式
val leaderStartOffset: Long = earliestOrLatestOffset(topicAndPartition,
    ListOffsetRequest.EARLIEST_TIMESTAMP, brokerConfig.brokerId)
// 选择下次获取消息的起始 offset
val offsetToFetch = Math.max(leaderStartOffset, replica.logEndOffset.
    messageOffset)
if (leaderStartOffset > replica.logEndOffset.messageOffset){
    // 将 Log 全部截断, 并创建新的 activeSegment
    replicaMgr.logManager.truncateFullyAndStartAt(topicAndPartition,
        leaderStartOffset)
}
offsetToFetch // 返回下次获取消息的 offset
```

`ReplicaFetcherThread` 还实现了 `handlePartitionsWithErrors()` 方法，它会对 `OFFSET_OUT_OF_RANGE` 之外的其他错误码做统一处理，其实现是将对应分区的同步操作暂停一段时间，即暂停一段其 `FetchRequest` 请求的发送。`handlePartitionsWithErrors()` 方法与 `buildFetchRequest()` 方法共同实现了此功能。

```
// 下面是 buildFetchRequest() 方法的实现
protected def buildFetchRequest(partitionMap:
    Map[TopicAndPartition, PartitionFetchState]): FetchRequest = {
    val requestMap = mutable.Map.empty[TopicPartition, JFetchRequest.
        PartitionData]
    partitionMap.foreach {
        case ((TopicAndPartition(topic, partition), partitionFetchState)) =>
            if (partitionFetchState.isActive) // 检测分区的同步状态是否为激活状态
                requestMap(new TopicPartition(topic, partition)) =
                    new JFetchRequest.PartitionData(partitionFetchState.offset,
                        fetchSize)
```



```

    }
    // 创建 FetchRequest
    new FetchRequest(new JFetchRequest(replicaId, maxWait, minBytes,
    requestMap.asJava))
}

// handlePartitionsWithErrors() 方法直接调用了 delayPartitions() 方法
def delayPartitions(partitions: Iterable[TopicAndPartition], delay: Long) {
    partitionMapLock.lockInterruptibly() // 加锁
    try {
        for (partition <- partitions) {
            partitionMap.get(partition).foreach (currentPartitionFetchState =>
                if (currentPartitionFetchState.isActive) // 检测分区的同步状态
                    // 将分区对应的同步状态由激活状态设置为延时状态, 延迟时长为 delay 毫秒
                    partitionMap.put(partition, new PartitionFetchState(
                        currentPartitionFetchState.offset, new DelayedItem(delay)))
        )
    }
    partitionMapCond.signalAll() // 唤醒 fetcher 线程
} finally partitionMapLock.unlock()
}

```

关闭副本

当 Broker 接收到来自 KafkaController 的 StopReplicaRequest 请求时, 会关闭其指定的副本, 并根据 StopReplicaRequest 中的字段决定是否删除副本对应的 Log。在分区的副本进行重新分配、关闭 Broker 等过程中都会使用到此请求, 但是需要注意的是, StopReplicaRequest 并不代表一定会删除副本对应的 Log, 例如 shutdown 的场景下就没有必要删除 Log。而在重新分配 Partition 副本的场景下, 就需要将旧副本及其 Log 删除。

先来介绍 StopReplicaRequest、StopReplicaResponse 的格式, 如图 4-53 所示。StopReplicaRequest 中的 delete_partitions 字段是一个 boolean 类型的值, 表示是否要删除副本及其 Log, partitions 集合字段中记录待关闭的分区信息。StopReplicaResponse 的 partitions 集合记录了每个分区对应的处理结果。

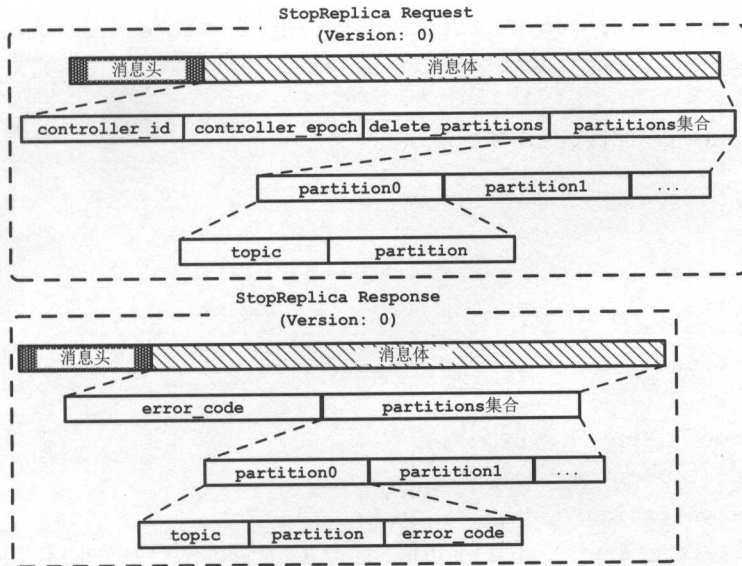


图 4-53

API 层接收到 `StopReplicaRequest` 后直接调用了 `ReplicaManager.stopReplicas()` 方法进行处理。`stopReplicas()` 方法首先检查请求中的 `controllerEpoch` 值，之后停止指定分区的同步操作，最后遍历 `partitions` 集合根据 `delete_partitions` 的值决定是否对 Log 进行删除。

```

def stopReplicas(stopReplicaRequest: StopReplicaRequest):
    (mutable.Map[TopicPartition, Short], Short) = {
        replicaStateChangeLock synchronized { // 加锁
            val responseMap = new collection.mutable.HashMap[TopicPartition,
Short]
            // 检查请求中的 controllerEpoch 值
            if (stopReplicaRequest.controllerEpoch() < controllerEpoch) {
                (responseMap, Errors.STALE_CONTROLLER_EPOCH.code)
            } else {
                val partitions = stopReplicaRequest.partitions.asScala
                controllerEpoch = stopReplicaRequest.controllerEpoch
                // 停止对指定分区的 fetch 操作
                replicaFetcherManager.removeFetcherForPartitions(partitions.map(r =>
TopicAndPartition(r.topic, r.partition)))
                for (topicPartition <- partitions) {
                    // 调用 stopReplica() 方法，关闭指定的分区的副本
                    val errorCode = stopReplica(topicPartition.topic,

```

```

        topicPartition.partition, stopReplicaRequest.deletePartitions)
    responseMap.put(topicPartition, errorCode)
  }
  (responseMap, Errors.NONE.code)
}
}
}

// 下面是 stopReplica() 方法的代码
def stopReplica(topic: String, partitionId: Int, deletePartition:
Boolean): Short = {
  val errorCode = Errors.NONE.code
  getPartition(topic, partitionId) match {
    case Some(partition) =>
      // 仅在 deletePartition 为 true 时, 才会真正删除该分区对应的副本及其 Log
      if (deletePartition) {
        val removedPartition = allPartitions.remove((topic, partitionId))
        if (removedPartition != null) {
          removedPartition.delete() // 删除副本, 这会导致 Log 被删除
        }
      }
      // allPartitions 中不存在对应的分区对象, 直接尝试删除 Log, 逻辑同上 (略)
      case None => ... ..
  }
  errorCode
}

```

ReplicaManager 中的定时任务

在 ReplicaManager 中 总 共 有 highwatermark-checkpoint、isr-expiration、isr-change-propagation 三个定时任务。highwatermark-checkpoint 任务会周期性地记录每个 Replica 的 HW 并保存到你 log 目录中的 replication-offset-checkpoint 文件中。isr-expiration 任务会周期性地调用 maybeShrinkIsr() 方法检测每个分区是否需要缩减其 ISR 集合。maybeShrinkIsr() 在前面已经介绍过了, 本节不再赘述。isr-change-propagation 任务会周期性地 将 ISR 集合发生变化的分区记录到 ZooKeeper 中。

前面分析 becomeLeaderOrFollower() 方法时提到过, 如果检测到 highwatermark-checkpoint 任务未启动, 会调用 startHighWaterMarksCheckPointThread() 方法启动 highwatermark-checkpoint

任务。

```
def startHighWaterMarksCheckPointThread() = {
  // 只能启动一次
  if (highWatermarkCheckPointThreadStarted.compareAndSet(false, true))
    scheduler.schedule("highwatermark-checkpoint", checkpointHighWatermarks,
      period = config.replicaHighWatermarkCheckpointIntervalMs,
      unit = TimeUnit.MILLISECONDS)
}

// 下面是 ReplicaManager.checkpointHighWatermarks() 方法
def checkpointHighWatermarks() {
  // 获取全部的 Replica 对象, 按照副本所在的 log 目录进行分组
  val replicas = allPartitions.values.flatMap(_.getReplica(config.brokerId))
  val replicasByDir = replicas.filter(_.log.isDefined)
    .groupBy(_.log.get.dir.getParentFile.getAbsolutePath)

  for ((dir, reps) <- replicasByDir) { // 遍历所有 log 目录
    // 收集当前 log 目录下的全部副本的 HW
    val hwms = reps.map(r => new TopicAndPartition(r) ->
      r.highWatermark.messageOffset).toMap
    try {
      // 使用 hwms 集合更新对应 log 目录下的 replication-offset-checkpoint 文件
      highWatermarkCheckpoints(dir).write(hwms)
    } catch { // 异常处理(略) }
  }
}
```

对 replication-offset-checkpoint 文件的读写由 OffsetCheckpoint 实现, 不再赘述。需要注意的是, 在 ReplicaManager 正常关闭的时候, 也会调用 checkpointHighWatermarks() 方法对 HW 进行一次记录。在 LogManager 中也有类似的做法, 在其关闭时会调用 checkpointLogsInDir() 方法对 RecoveryPointCheckpoint 文件进行一次更新。

再来看 isr-expiration 和 isr-change-propagation 这两个定时任务, 它们是在 ReplicaManager 的 startup() 方法中启动的。


```
def startup() {
    scheduler.schedule("isr-expiration", maybeShrinkIsr,
        period = config.replicaLagTimeMaxMs, unit = TimeUnit.MILLISECONDS)
    scheduler.schedule("isr-change-propagation", maybePropagateIsrChanges,
        period = 2500L, unit = TimeUnit.MILLISECONDS)
}
```

在 `Partition.maybeShrinkIsr()` 方法和 `maybeExpandIsr()` 方法中都会调用 `updateIsr()` 方法更新 `inSyncReplicas` 字段，并将新 ISR 集合上传到 ZK 中保存，同时还会调用 `recordIsrChanger()` 方法记录 ISR 记录发生变化的分区。

```
def recordIsrChange(topicAndPartition: TopicAndPartition) {
    isrChangeSet synchronized { // 加锁
        isrChangeSet += topicAndPartition // 添加到 isrChangerSet
        lastIsrChangeMs.set(System.currentTimeMillis()) // 记录最后更新时间
    }
}
```

`isr-change-propagation` 任务会定期将 ISR 集合发送变化的分区记录到 ZooKeeper 中。KafkaController 对相应路径添加了 Watcher，当 Watcher 被触发后会向其管理的 Broker 发送 `UpdateMetadataRequest`，频繁地触发 Watcher 会影响 KafkaController、ZooKeeper 甚至其他 Broker 的性能。为避免这种情况，`maybePropagateIsrChanges()` 方法设置了一定的写入条件：`isrChangeSet` 集合不为空且最后一次有 ISR 集合发生变化的时间距今已超过 5 秒或者上次写入 ZooKeeper 的时间距今已超过 60 秒。

```
def maybePropagateIsrChanges() {
    val now = System.currentTimeMillis()
    isrChangeSet synchronized { // 加锁
        // 下面的判断条件依次是：isrChangeSet 不为空，最后一次有 ISR 集合发生变化的时间距今已超过 5 秒
        // 上次写入 ZooKeeper 的时间距今已超过 60 秒
        if (isrChangeSet.nonEmpty &&
            (lastIsrChangeMs.get() + ReplicaManager.IsrChangePropagationBlackOut
            < now ||
            lastIsrPropagationMs.get() + ReplicaManager.IsrChangePropagationInterval
            < now)) {

            // 将 isrChangeSet 写入 ZooKeeper
        }
    }
}
```

```

ReplicationUtils.propagateIsrChanges(zkUtils, isrChangeSet)
isrChangeSet.clear() // 清空 isrChangeSet
lastIsrPropagationMs.set(now) // 更新 lastIsrPropagationMs
}
}
}

```

MetadataCache

MetadataCache 是 Broker 用来缓存整个集群中全部分区状态的组件。KafkaController 通过向集群中的 Broker 发送 UpdateMetadataRequest 来更新其 MetadataCache 中缓存的数据，每个 Broker 在收到该请求后会异步更新 MetadataCache 中的数据。

MetadataCache 中各字段的含义和功能如下所述。

- **cache:** Map[String, Map[Int, PartitionStateInfo]] 类型，记录了每个分区的状态，其中使用 PartitionStateInfo 记录 Partition 的状态。

```

case class PartitionStateInfo(leaderIsrAndControllerEpoch:
    LeaderIsrAndControllerEpoch, allReplicas: Set[Int]) {...}

case class LeaderIsrAndControllerEpoch(leaderAndIsr: LeaderAndIsr,
    controllerEpoch: Int) {...}

case class LeaderAndIsr(var leader: Int, var leaderEpoch: Int, var isr:
    List[Int], var zkVersion: Int) {...}

```

PartitionStateInfo 中记录了 AR 集合、ISR 集合、Leader 副本的 id、leaderEpoch 和 controllerEpoch。

- **aliveBrokers:** Map[Int, Broker] 类型，记录了当前可用的 Broker 信息，其中使用 Broker 类记录每个存活 Broker 的网络位置信息（host、ip、port 等）。
- **aliveNodes:** Map[Int, Map[SecurityProtocol, Node]] 类型，记录了可用节点的信息。

在开始分析 Kafka 处理 UpdateMetadataRequest 请求更新 MetadataCache 的流程之前，先来看一下 UpdateMetadataRequest 和 UpdateMetadataResponse 的格式，如图 4-54 所示。

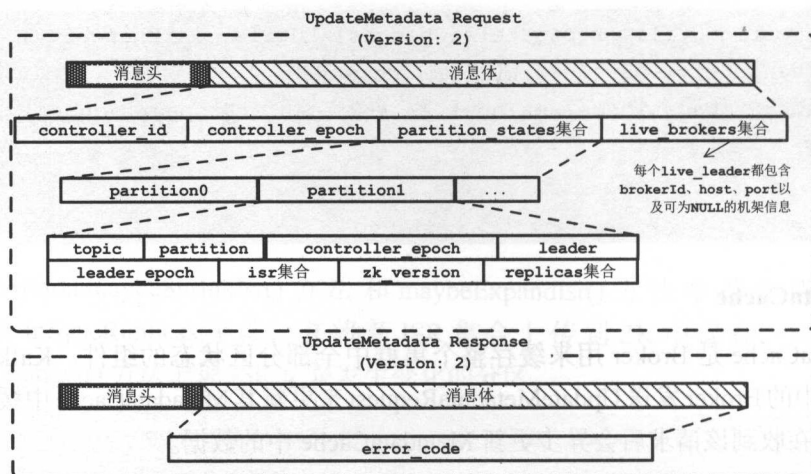


图 4-54

UpdateMetadataRequest 由 KafkaApis.handleUpdateMetadataRequest() 方法处理，它直接将请求交给 ReplicaManager.maybeUpdateMetadataCache() 方法处理。

```
def maybeUpdateMetadataCache(correlationId: Int,
    updateMetadataRequest: UpdateMetadataRequest, metadataCache:
MetadataCache) {
    replicaStateChangeLock synchronized {
        // 检查 controllerEpoch
        if (updateMetadataRequest.controllerEpoch < controllerEpoch) {
            ..... // 异常处理 (略)
        } else {
            // 更新 MetadataCache
            metadataCache.updateCache(correlationId, updateMetadataRequest)
            controllerEpoch = updateMetadataRequest.controllerEpoch // 更新
        }
    }
}
```

MetadataCache.updateCache() 方法中完成了对 aliveBrokers、cache 等字段的更新，相关代码如下：


```

def updateCache(correlationId: Int, updateMetadataRequest: UpdateMetadataRequest) {
  inWriteLock(partitionMetadataLock) { // 加锁
    controllerId = updateMetadataRequest.controllerId match { // 更新
      controllerId
        case id if id < 0 => None
        case id           => Some(id)
    }

    // 将 aliveNodes 和 aliveBrokers 中的缓存全部清除掉，并从 UpdateMetadataRequest
    // 中的 live_borkers 字段重新构建
    aliveNodes.clear()
    aliveBrokers.clear()
    updateMetadataRequest.liveBrokers.asScala.foreach { broker =>
      ...
      aliveBrokers(broker.id) = Broker(broker.id,
        *   endPoints.asScala, Option(broker.rack))
      aliveNodes(broker.id) = nodes.asScala
    }

    // 根据 UpdateMetadataRequest.partition_states 字段更新 cache 集合
    updateMetadataRequest.partitionStates.asScala.foreach {
      case (tp, info) =>
        val controllerId = updateMetadataRequest.controllerId
        val controllerEpoch = updateMetadataRequest.controllerEpoch
        if (info.leader == LeaderAndIsr.LeaderDuringDelete) {
          // 删除分区对应的 PartitionStateInfo
          removePartitionInfo(tp.topic, tp.partition)
        } else { // 更新 PartitionStateInfo
          val partitionInfo = partitionStateToPartitionStateInfo(info)
          addOrUpdatePartitionInfo(tp.topic, tp.partition, partitionInfo)
        }
    }
  }
}

```

在之前分析的生产者和消费者中使用 Metadata 对象缓存 Kafka 集群的元信息，在 Metadata 更新时会向服务端发送 MetadataRequest。MetadataRequest 首先由 KafkaApis.handleTopicMetadataRequest() 方法进行处理，其实现如下：


```

def handleTopicMetadataRequest(request: RequestChannel.Request) {
    val metadataRequest = request.body.asInstanceOf[MetadataRequest]
    val requestVersion = request.header.apiVersion()
    .....// 对 MetadataRequest (Version 0) 的处理 (略)
    val topics = // 如果请求的 topic 集合字段为空集合, 则读取所有的 Topic 信息
        if (metadataRequest.isAllTopics) metadataCache.getAllTopics()
        else metadataRequest.topics.asScala.toSet
    }
    .....
    // 权限验证, 设置未验证通过的 topic 集合的错误码为 TOPIC_AUTHORIZATION_FAILED (略)

    // 查询 MetadataCache 得到指定 Topic 的信息
    val topicMetadata = getTopicMetadata(authorizedTopics, request.
securityProtocol,
        errorUnavailableEndpoints)

    // 按照 MetadataResponse 格式创建响应 (略)
    // 向 RequestChannel 中添加响应
    requestChannel.sendResponse(new RequestChannel.Response(request,
        new ResponseSend(request.connectionId, responseHeader,
responseBody)))
}

```

在 `KafkaApis.getTopicMetadata()` 方法中完成对 `MetadataCache` 的查询, 同时还会根据配置以及 Topic 的名称决定是否自动创建未知 (`MetadataCache` 查找不到) 的 Topic。

```

private def getTopicMetadata(topics: Set[String]...):
Seq[MetadataResponse.TopicMetadata] = {
    // 查询 MetadataCache
    val topicResponses = metadataCache.getTopicMetadata(topics,
securityProtocol, errorUnavailableEndpoints)
    if (topics.isEmpty || topicResponses.size == topics.size) {
        topicResponses // MetadataCache 中可以找到全部指定的 Topic
    } else {
        .....// 根据配置决定是否调用 createTopic() 方法创建未知的 Topic (略)
    }
}

```

在 `MetadataCache.getTopicMetadata()` 方法中会查询 `cache` 字段以获取指定 Topic 的信息。

```
def getTopicMetadata(topics: Set[String], protocol: SecurityProtocol,
    errorUnavailableEndpoints: Boolean = false): Seq[MetadataResponse.
TopicMetadata] = {
    inReadLock(partitionMetadataLock) { // 加锁
        topics.toSeq.flatMap { topic =>
            // 调用 getPartitionMetadata() 方法获取 PartitionMetadata, 并将
            // PartitionMetadata 与 Topic 信息进行封装, 生成 TopicMetadata
            getPartitionMetadata(topic, protocol, errorUnavailableEndpoints)
                .map { partitionMetadata =>
                    new MetadataResponse.TopicMetadata(Errors.NONE, topic,
                        Topic.isInternal(topic), partitionMetadata.toBuffer.
asJava)
                }
        }
    }
}

// 下面是 MetadataCache.getPartitionMetadata() 方法的实现
private def getPartitionMetadata(topic: String, protocol: SecurityProtocol,
    errorUnavailableEndpoints: Boolean):
    Option[Iterable[MetadataResponse.PartitionMetadata]] = {
    // 获取每个 Topic 的分区集合, 并对其进行遍历
    cache.get(topic).map { partitions =>
        partitions.map {
            case (partitionId, partitionState) =>
                val topicPartition = TopicAndPartition(topic, partitionId)
                // 获取分区的对应的 leaderAndIsr 对象, 其中记录了 leaderId、leaderEpoch、
                // ISR 集合以及 controllerEpoch
                val leaderAndIsr = partitionState.leaderIsrAndControllerEpoch.
leaderAndIsr
                // 获取 Leader 副本所在的 Node, 其中记录了 host、ip、port
                val maybeLeader = getAliveEndpoint(leaderAndIsr.leader, protocol)
                val replicas = partitionState.allReplicas // 获取分区的 AR 集合
                // 获取 AR 集合中可用的副本
                val replicaInfo = getEndpoints(replicas, protocol,
errorUnavailableEndpoints)
```

```

maybeLeader match {
    // 分区的 Leader 副本可能宕机了, 错误码为 LEADER_NOT_AVAILABLE (略)
    case None =>... ..
    case Some(leader) =>
        val isr = leaderAndIsr.isr // 获取分区的 ISR 集合
        // 获取 ISR 集合中可用的副本
        val isrInfo = getEndpoints(isr, protocol, errorUnavailableEndpoints)
        // 检测 AR 集合中的副本是否都是可用的
        if (replicaInfo.size < replicas.size) {
            new MetadataResponse.PartitionMetadata(Errors.REPLICA_NOT_
AVAILABLE,
                partitionId, leader, replicaInfo.asJava, isrInfo.asJava)
            // 检测 ISR 集合中的副本是否都是可用的
        } else if (isrInfo.size < isr.size) {
            new MetadataResponse.PartitionMetadata(Errors.REPLICA_NOT_
AVAILABLE,
                partitionId, leader, replicaInfo.asJava, isrInfo.asJava)
        } else { // AR 和 ISR 集合的副本都是可用的
            new MetadataResponse.PartitionMetadata(Errors.NONE,
partitionId, leader,
                replicaInfo.asJava, isrInfo.asJava)
        }
    }
}
}
}
}

```

MetadataCache 中还拥有了很多 get*() 方法, 它们都是从上述三个集合字段中获取缓存数据, 不再赘述了, 感兴趣的读者请参考源码学习。

本节主要介绍了 Kafka 中副本机制的相关概念和实现。首先介绍了 Replica 中维护的副本信息以及“Local Replica”和“Remote Replica”的概念, 然后介绍了 Partition 如何管理整个 Broker 范围内的副本、切换副本角色、管理 ISR 集合、追加消息等功能。最后, 介绍了 ReplicaManager 中提供的同步功能, 管理 MetadataCache 的相关功能, 管理 Partition 的功能。希望读者阅读完本节后, 能够对 Kafka 的副本机制有比较清晰的了解。

4.6 KafkaController

在上一节对副本机制的实现进行了分析，其中提到 Broker 能够处理来自 KafkaController 的 LeaderAndIsrRequest、StopReplicaRequest、UpdateMetadataRequest 等请求。本节将介绍 KafkaController 在集群中扮演的角色以及 KafkaController 与各个 Broker 之间如何协同工作。

在 Kafka 集群的多个 Broker 中，有一个 Broker 会被选举为 Controller Leader，负责管理整个集群中所有的分区和副本的状态。例如：当某分区的 Leader 副本出现故障时，由 Controller 负责为该分区重新选举新的 Leader 副本；当使用 kafka-topics 脚本增加某 Topic 的分区数量时，由 Controller 管理分区的重新分配；当检测到分区的 ISR 集合发生变化时，由 Controller 通知集群中所有的 Broker 更新其 MetadataCache 信息。

为了实现 Controller 的高可用，一个 Broker 被选为 Leader 之后，其他的 Broker 都会成为 Follower（不加特殊说明的情况下，本节的“Leader/Follower”指的都是 KafkaController 的 Leader/Follower，请读者不要与副本机制中的 Leader 副本和 Follower 副本混淆），会从剩下的 Follower 中选出新的 Controller Leader 来管理集群。

选举 Controller Leader 依赖于 ZooKeeper 实现，每个 Broker 启动时都会创建一个 KafkaController 对象，但是集群中只能存在一个 Controller Leader 来对外提供服务。在集群启动时，多个 Broker 上的 KafkaController 会在指定路径下竞争创建节点，只有第一个成功创建节点的 KafkaController 才能成为 Leader，而其余的 KafkaController 则成为 Follower。当 Leader 出现故障后，所有的 Follower 会收到通知，再次竞争在该路径下创建节点从而选出新的 Leader。这也是 ZooKeeper 的一种常见用法。

在 Kafka 早期版本中并没有采用 KafkaController 的设计来对分区和副本状态进行管理，而是依赖于 ZooKeeper 的 Watcher 和队列。在早期版本的设计中，每个 Broker 都会在 ZooKeeper 上注册 Watcher，ZooKeeper 上就会出现大量 Watcher，当分区或副本状态变化时会唤醒很多不必要的 Watcher，这种严重依赖 ZooKeeper 的设计出现了“脑裂”、“羊群效应”以及 ZooKeeper 集群过载的情况。在新版本设计中，只有 Controller Leader 在 ZooKeeper 上注册 Watcher，其他 Broker 几乎不用再监听 ZooKeeper 中的数据变化。旧版本中 Broker 之间传递事件依赖于 ZooKeeper 的设计比较低效，在新版设计中 Controller Leader 直接与 Broker 交互。旧版本的设计毕竟已经废弃，本节不做过多介绍。需要读者了解的是，在设计分布式系统时要适度依赖 ZooKeeper 集群，合理利用 ZooKeeper Watcher，否则就会出现上述问题。

我们先通过图 4-55 了解 ZooKeeper 中与 KafkaController 相关的路径以及该路径中记

录的内容的含义。

- `/brokers/ids/[id]`: 记录了集群中可用 Broker 的 id。
- `/brokers/topics/[topic]/partitions`: 记录了一个 Topic 中所有分区的分配信息以及 AR 集合信息。
- `/brokers/topics/[topic]/partitions/[partition_id]/state`: 记录了某 Partition 的 Leader 副本所在 BrokerId、lead_epoch、ISR 集合、ZKVersion 等信息。
- `/controller_epoch`: 记录了当前 Controller Leader 的年代信息。
- `/controller`: 记录了当前 Controller Leader 的 Id, 也用于 Controller Leader 的选举。
- `/admin/reassign_partitions`: 记录了需要进行副本重新分配的分区。
- `/admin/preferred_replica_election`: 记录了需要进行“优先副本”选举的分区。“优先副本”是在创建分区时为其指定的第一个副本。
- `/admin/delete_topics`: 记录了待删除的 Topic。
- `/isr_change_notification`: 记录了一段时间内 ISR 集合发生变化的分区。
- `/config`: 记录了一些配置信息。

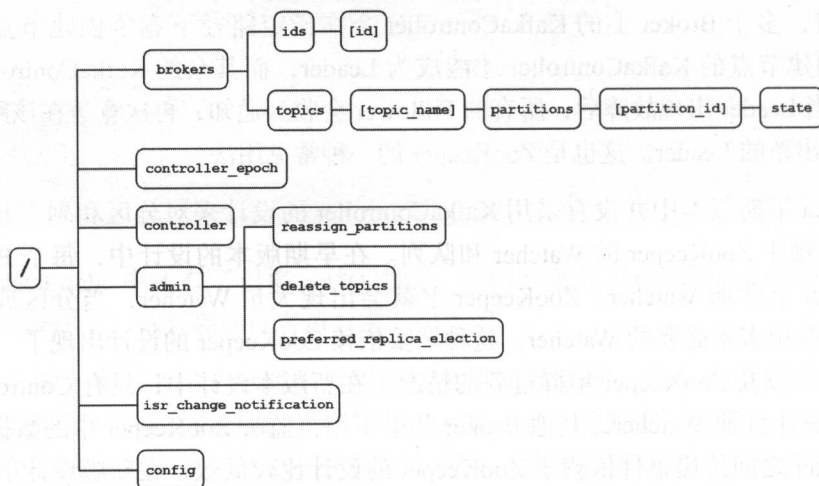


图 4-55

在详细介绍 KafkaController 的相关组件之前, 先从整体上了解 KafkaController 的设计, 以及组件之间的依赖关系如图 4-56 所示。

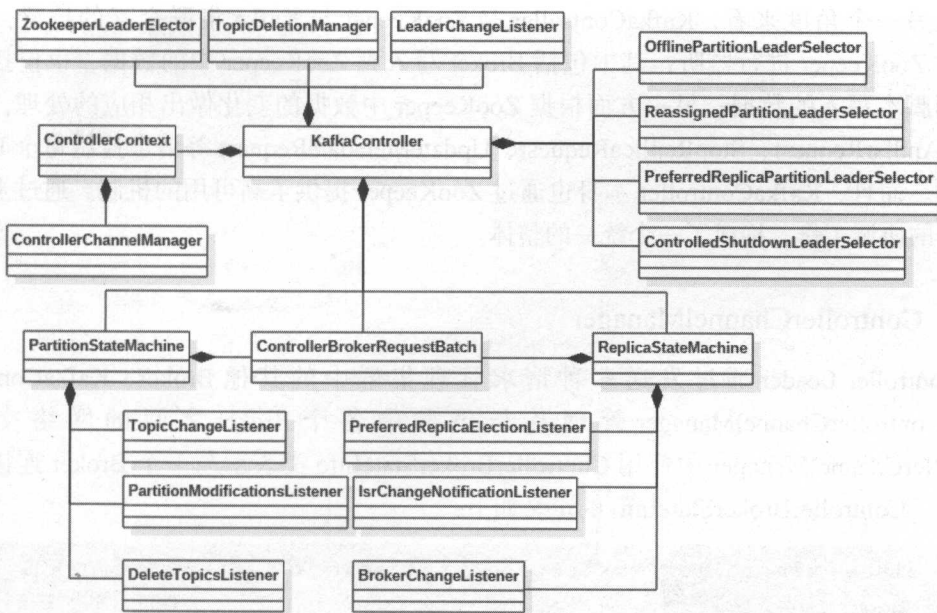


图 4-56

- **KafkaController** 组织并封装了其他组件，对外提供 API 接口。
- **ZookeeperLeaderElector** 主要用于 Controller Leader 的选举。
- **ControllerContext** 是 **KafkaController** 的上下文信息，缓存了 ZooKeeper 中记录的整个集群的元信息，例如，可用 Broker、全部的 Topic、分区、副本的信息。
- **ControllerChannelManager** 维护了 Controller Leader 与集群中其他 Broker 之间的网络连接，是管理整个集群的基础。
- **TopicDeletionManager** 用于对指定的 Topic 进行删除。
- **PartitionStateMachine** 用于管理集群中所有 Partition 状态的状态机。
- **ReplicaStateMachine** 用于管理集群中所有副本状态的状态机。
- **ControllerBrokerRequestBatch** 实现了向 Broker 批量发送请求的功能。
- ***PartitionLeaderSelector** 实现了多种 Leader 副本选举策略。
- ***Listener** 是 ZooKeeper 上的监听器，实现了对 ZooKeeper 上某些节点中的数据、子节点或 ZooKeeper Session 状态的监听，被触发后调用相应的业务逻辑。

从另一个角度来看, `KafkaController` 是 `ZooKeeper` 与 `Kafka` 集群交互的桥梁: 它一方面对 `ZooKeeper` 进行监听, 其中包括 `Broker` 写入到 `ZooKeeper` 中的数据, 也包括管理员使用脚本写入的数据; 另一方面根据 `ZooKeeper` 中数据的变化做出相应的处理, 通过 `LeaderAndIsrRequest`、`StopReplicaRequest`、`UpdateMetadataRequest` 等请求控制每个 `Broker` 的工作。而且, `KafkaController` 本身也通过 `ZooKeeper` 提供了高可用的机制。通过上述组件之间的协调工作, 构成了一个统一的整体。

4.6.1 ControllerChannelManager

`Controller Leader` 通过发送多种请求管理集群中的其他 `Broker`, `KafkaController` 使用 `ControllerChannelManager` 管理其与集群中各个 `Broker` 之间的网络交互。`ControllerChannelManager` 中使用 `ControllerBrokerStateInfo` 类表示与一个 `Broker` 连接的各种信息。`ControllerBrokerStateInfo` 的定义如下:

```
case class ControllerBrokerStateInfo(networkClient: NetworkClient,
brokerNode: Node,
    messageQueue: BlockingQueue[QueueItem], requestSendThread:
RequestSendThread)
```

其 `networkClient` 字段负责维护 `Controller` 与对应 `Broker` 通信的网络连接, 这里还是与 `NetworkClientBlockingOps` 配合实现阻塞的使用方式。`brokerNode` 字段维护了对应的 `Broker` 的网络位置信息, 其中记录了 `Broker` 的 `host`、`ip`、`port` 以及机架信息。`requestSendThread` 字段是用于发送请求的线程。`messageQueue` 字段是一个缓冲队列 (`LinkedBlockingQueue` 类型), 存放了发往对应 `Broker` 的请求, 其中每个元素是 `QueueItem` 类型, 其中封装了 `Request` 本身和其对应的回调函数, 其定义如下:

```
case class QueueItem(apiKey: ApiKeys, apiVersion: Option[Short],
    request: AbstractRequest, callback: AbstractRequestResponse =>
Unit)
```

`RequestSendThread` 继承了 `ShutdownableThread`, 在线程停止之前会循环执行 `doWork()` 方法, 通过 `NetworkClientBlockingOps` 完成发送请求并阻塞等待响应。

```

override def doWork(): Unit = {
  // 获取缓冲队列中的 QueueItem
  val QueueItem(apiKey, apiVersion, request, callback) = queue.take()
  import NetworkClientBlockingOps._
  var clientResponse: ClientResponse = null
  try {
    lock synchronized {
      var isSendSuccessful = false
      while (isRunning.get() && !isSendSuccessful) {
        // 当 Broker 宕机后, 会触发 ZooKeeper 的监听器调用 removeBroker() 方法将当前
        // 线程停止, 在停止前会一直尝试重试
        try {
          if (!brokerReady()) { // 阻塞等待符合发送条件
            isSendSuccessful = false
            backoff() // 退避一段时间, 三秒
          } else {
            ... .. // 创建 ClientRequest 对象 (略)

            // 发送请求并阻塞等待响应
            clientResponse = networkClient.blockingSendAndReceive(clientRe
quest)(time)
            isSendSuccessful = true // 设置发送成功的标记
          }
        } catch {
          // 异常处理, 如果发送失败, 则将连接断开, 退避一段时间后重连再尝试发送 (略)
          ... ..
        }
      }
    }
    if (clientResponse != null) {
      // 检测请求类型, Controller 只能发送 LeaderAndIsrRequest、StopReplicaRequest、
      // UpdateMetadataRequest 三种请求 (略)
      // 调用 QueueItem 中封装的回调函数
      if (callback != null) callback(response)
    }
  } catch {
    ... .. // 异常处理, 断开连接 (略)
  }
}

```


ControllerChannelManager 的核心字段是 brokerStateInfo (HashMap[Int, ControllerBrokerStateInfo] 类型)，用于管理集群中每个 Broker 对应的 ControllerBrokerStateInfo 对象。其初始化过程如下：

```
// 为 Broker 创建对应的 ControllerBrokerStateInfo 对象
controllerContext.liveBrokers.foreach(addNewBroker(_))

def startup() = {
  brokerLock synchronized {
    // 启动 RequestSendThread 线程
    brokerStateInfo.foreach(brokerState => startRequestSendThread(brokerState._1))
  }
}
```

ControllerChannelManager.addNewBroker() 方法和 removeBroker() 方法实现了对 brokerStateInfo 集合的管理，sendRequest() 方法向指定 Broker 发送请求。

```
private def addNewBroker(broker: Broker) {
  val messageQueue = new LinkedBlockingQueue[QueueItem] // 创建消息队列
  val brokerEndPoint = broker.getBrokerEndPoint(config.interBrokerSecurityProtocol)
  val brokerNode = new Node(broker.id, brokerEndPoint.host, brokerEndPoint.port)
  val networkClient = {
    .....// 创建 NetworkClient 对象的过程 (略)
  }
  val requestThread = new RequestSendThread(...)
  // 填充 brokerStateInfo 集合
  brokerStateInfo.put(broker.id, new ControllerBrokerStateInfo(networkClient,
    brokerNode, messageQueue, requestThread))
}

private def removeExistingBroker(brokerState: ControllerBrokerStateInfo) {
  try {
    brokerState.networkClient.close() // 关闭底层连接
    brokerState.messageQueue.clear() // 清空队列
    brokerState.requestSendThread.shutdown() // 关闭 RequestSendThread
  }
```

```

// 清除 ControllerBrokerStateInfo 对象
brokerStateInfo.remove(brokerState.brokerNode.id)
} catch {..... // 异常处理(略)}
}

def sendRequest(brokerId: Int, apiKey: ApiKeys, apiVersion: Option[Short],
  request: AbstractRequest, callback: AbstractRequestResponse => Unit =
  null) {
  brokerLock synchronized {
    val stateInfoOpt = brokerStateInfo.get(brokerId)
    stateInfoOpt match {
      case Some(stateInfo) =>
        // 很明显是放到一个队列里面缓存了
        stateInfo.messageQueue.put(QueueItem(apiKey, apiVersion, request,
        callback))
      case None => ..... // 输出警告信息
    }
  }
}

```

4.6.2 ControllerContext

ControllerContext 中维护了 Controller 使用到的上下文信息，从其构造函数也能猜到，ControllerContext 与 ZooKeeper 有密切的关系，也可以将 ControllerContext 看作 ZooKeeper 数据的缓存。

```
class ControllerContext(val zkUtils: ZkUtils, val zkSessionTimeout: Int) ...
```

ControllerContext 中各个字段的含义和作用如下所述。

- controllerChannelManager: 管理 Controller 与集群中 Broker 之间的连接。
- shuttingDownBrokerIds: 正在关闭的 BrokerId 集合。
- epoch: Controller 的年代信息，初始为 0。Controller 的年代信息存储的 ZK 路径是 “/controller_epoch”。每次重新选举新的 Leader Controller，epoch 字段值就会增加 1。
- epochZkVersion: 年代信息的 ZK 版本，初始为 0。
- allTopics: 整个集群中全部的 Topic 名称。

- `partitionReplicaAssignment`: `Map[TopicAndPartition, Seq[Int]]` 类型, 记录了每个分区的 AR 集合。
- `partitionLeadershipInfo`: `Map[TopicAndPartition, LeaderIsrAndControllerEpoch]` 类型, 记录了每个分区的 Leader 副本所在的 `BrokerId`、ISR 集合以及 `controller_epoch` 等信息。其中 `LeaderIsrAndControllerEpoch` 的定义如下:

```
case class LeaderIsrAndControllerEpoch(leaderAndIsr: LeaderAndIsr,
    controllerEpoch: Int)

case class LeaderAndIsr(var leader: Int, var leaderEpoch: Int,
    var isr: List[Int], var zkVersion: Int) {
```

- `partitionBeingReassigned`: `Map[TopicAndPartition, ReassignedPartitionsContext]` 类型, 记录了正在重新分配副本的分区。该集合的 value 是 `ReassignedPartitionsContext` 类型, 其中封装了新分配的 AR 集合信息以及用于监听 ISR 集合变化的 `ReassignedPartitionsIsrChangeListener`, 其定义如下:

```
case class ReassignedPartitionsContext(var newReplicas: Seq[Int] = Seq.empty,
    var isrChangeListener: ReassignedPartitionsIsrChangeListener = null)
```

- `partitionsUndergoingPreferredReplicaElection`: `Set[TopicAndPartition]` 类型, 记录了正在进行“优先副本”选举的分区。
- `liveBrokersUnderlying`: `Set[Broker]` 类型, 记录了当前可用的 Broker 集合。
- `liveBrokerIdsUnderlying`: `Set[Int]` 类型, 记录了当前可用的 `BrokerId` 集合。

`ControllerContext` 为 `liveBrokersUnderlying` 字段、`liveBrokerIdsUnderlying` 字段和 `shuttingDownBrokerIds` 字段提供了相关的集合操纵方法。

- `liveBrokers_`: 根据提供的 Broker 集合对 `liveBrokersUnderlying` 集合和 `liveBrokerIdsUnderlying` 集合进行更新。
- `liveBrokers/liveBrokerIds`: 从 `liveBrokersUnderlying/liveBrokerIdsUnderlying` 集合中排除 `shuttingDownBrokerIds` 集合后返回。
- `liveOrShuttingDownBrokerIds/liveOrShuttingDownBrokers`: 获取 `liveBrokersUnderlying/liveBrokerIdsUnderlying` 集合。

ControllerContext 为 partitionReplicaAssignment 字段提供的管理方法如下所述。

- partitionsOnBroker: 获取在指定 Broker 中存在有副本的分区集合。
- replicasOnBrokers: 获取指定 Broker 集合中保存的所有副本。
- replicasForTopic: 获取指定 Topic 的所有副本。
- partitionsForTopic: 获取指定 Topic 的所有分区。
- allLiveReplicas: 获取所有可用 Broker 中保存的副本。
- replicasForPartition: 获取指定分区集合的副本。
- removeTopic: 删除指定 Topic。

这些方法的代码比较简单, 请读者参考源码学习。ControllerContext 中各个集合填充数据的逻辑比较零散, 后面分析中遇到时会做说明。

4.6.3 ControllerBrokerRequestBatch

为了提高 Controller Leader 与集群中其他 Broker 的通信效率, KafkaController 使用 ControllerBrokerRequestBatch 组件实现批量发送请求的功能。

ControllerBrokerRequestBatch 的核心字段如下所述。

- leaderAndIsrRequestMap: Map [Int, Map[TopicPartition, PartitionStateInfo]] 类型, 记录了发往指定 Broker 的 LeaderAndIsrRequest 所需的信息, 其中 PartitionStateInfo 的定义如下:

```
case class PartitionStateInfo(
  leaderIsrAndControllerEpoch: LeaderIsrAndControllerEpoch,
  allReplicas: Set[Int])
```

- stopReplicaRequestMap: Map[Int, Seq[StopReplicaRequestInfo]] 类型, 记录了发往指定 Broker 的 StopReplicaRequest 所需的信息, 其中 StopReplicaRequestInfo 的定义如下:

```
case class StopReplicaRequestInfo(replica: PartitionAndReplica,
  deletePartition: Boolean, callback: AbstractRequestResponse => Unit =
  null)
```

- updateMetadataRequestMap: Map [Int, Map[TopicPartition, PartitionStateInfo]] 类型,

记录了发往指定 Broker 的 UpdateMetadataRequest 集合。

ControllerBrokerRequestBatch 的常规用法如下：

```
try {
    brokerRequestBatch.newBatch()
    brokerRequestBatch.add*RequestForBrokers(...)
    brokerRequestBatch.sendRequestsToBrokers(epoch)
} catch {
    brokerRequestBatch.clear()
}
```

ControllerBrokerRequestBatch.newBatch() 方法会检测三个请求集合是否为空，如果不为空则抛出异常。ControllerBrokerRequestBatch.clear() 方法则会清空三个请求集合。

ControllerBrokerRequestBatch.addLeaderAndIsrRequestForBrokers() 方法会向 leaderAndIsrRequestMap 集合中添加待发送的 LeaderAndIsrRequest 所需的数据，同时会调用 addUpdateMetadataRequestForBrokers() 方法准备向集群中所有可用的 Broker 发送 UpdateMetadataRequest。

```
// 第一个参数指定了接收 LeaderAndIsrRequest 的 Broker 集合
def addLeaderAndIsrRequestForBrokers(brokerIds: Seq[Int], topic: String,
    partition: Int, leaderIsrAndControllerEpoch: LeaderIsrAndControllerEpoch,
    replicas: Seq[Int], callback: AbstractRequestResponse => Unit = null) {

    val topicPartition = new TopicPartition(topic, partition)
    // 查找 Broker 对应的集合
    brokerIds.filter(_ >= 0).foreach { brokerId =>
        val result = leaderAndIsrRequestMap.getOrElseUpdate(brokerId, mutable.
Map.empty)
        // 添加 Leader、ISR、AR 等构造 LeaderAndIsrRequest 请求需要的信息
        result.put(topicPartition, PartitionStateInfo(leaderIsrAndControllerEpoch,
            replicas.toSet))
    }
    // 准备向所有可用的 Broker 发送 UpdateMetadataRequest
    addUpdateMetadataRequestForBrokers(controllerContext.liveOrShuttingDownBrokerIds
        .toSeq, Set(TopicAndPartition(topic, partition)))
}
```

`ControllerBrokerRequestBatch.addUpdateMetadataRequestForBrokers()` 方法的代码如下:

```
def addUpdateMetadataRequestForBrokers(brokerIds: Seq[Int],
    partitions: collection.Set[TopicAndPartition] = Set.empty[TopicAndPartition],
    callback: AbstractRequestResponse => Unit = null) {
    // 定义回调函数
    def updateMetadataRequestMapFor(partition: TopicAndPartition,
        beingDeleted: Boolean)
    {
        // 首先找出 controller 中保存的该分区的 leader
        val leaderIsrAndControllerEpochOpt =
            controllerContext.partitionLeadershipInfo.get(partition)
        leaderIsrAndControllerEpochOpt match {
            // 获取分区的 AR 集合
            val replicas = controllerContext.partitionReplicaAssignment(partition)
            .toSet
            case Some(leaderIsrAndControllerEpoch) =>
                val partitionStateInfo = if (beingDeleted) {
                    // 根据 beingDeleted 参数设置 leader 的值
                    val leaderAndIsr = new LeaderAndIsr(LeaderAndIsr.LeaderDuringDelete,
                        leaderIsrAndControllerEpoch.leaderAndIsr.isr)
                    PartitionStateInfo(LeaderIsrAndControllerEpoch(leaderAndIsr,
                        leaderIsrAndControllerEpoch.controllerEpoch), replicas)
                } else {
                    PartitionStateInfo(leaderIsrAndControllerEpoch, replicas)
                }
            // 向 updateMetadataRequestMap 中添加数据
            brokerIds.filter(b => b >= 0).foreach { brokerId =>
                updateMetadataRequestMap.getOrElseUpdate(brokerId,
                    mutable.Map.empty[TopicPartition, PartitionStateInfo])
                updateMetadataRequestMap(brokerId).put(new TopicPartition(partition.
                    topic,
                        partition.partition), partitionStateInfo)
            }
            case None => ... ..// 日志输出 (略)
        }
    }
}
```

```

// 如果指定的分区集合为空，则需要更新全部分区
val filteredPartitions = {
    val givenPartitions = if (partitions.isEmpty)
        controllerContext.partitionLeadershipInfo.keySet
    else
        partitions
    // 过滤即将被删除的 Partition
    if (controller.deleteTopicManager.partitionsToBeDeleted.isEmpty)
        givenPartitions
    else
        givenPartitions -- controller.deleteTopicManager.partitionsToBeDeleted
}
... ..// 边界检查 (略)
// 将 filteredPartitions 中的分区信息添加到 updateMetadataRequestMap 集合中，等待发送
filteredPartitions.foreach(
    partition => updateMetadataRequestMapFor(partition, beingDeleted =
false))
// 将即将被删除的分区信息添加到 updateMetadataRequestMap 集合中，等待发送
// 注意 beingDeleted 参数
controller.deleteTopicManager.partitionsToBeDeleted.foreach(
    partition => updateMetadataRequestMapFor(partition, beingDeleted =
true))
}

```

`addStopReplicaRequestForBrokers()` 方法会向 `stopReplicaRequestMap` 集合中添加 `StopReplicaRequest` 所需的数据，具体实现与上述两个 `add*RequestForBroker()` 类似，不再赘述。

`ControllerBrokerRequestBatch.sendRequestsToBrokers()` 方法会使用上述三个集合中的数据来创建相应的请求，并添加到 `ControllerChannelManager` 中对应的 `messageQueue` 队列中，最终由 `RequestSendThread` 线程将请求发送出去。


```

def sendRequestsToBrokers(controllerEpoch: Int) {
  // 第一步: 处理 leaderAndIsrRequestMap 集合
  leaderAndIsrRequestMap.foreach {
    case (broker, partitionStateInfos) =>
      .....// 根据 leaderAndIsrRequestMap 集合创建 LeaderAndIsrRequest 对象
      val leaderAndIsrRequest = new LeaderAndIsrRequest(controllerId,
        controllerEpoch, partitionStates.asJava, leaders.asJava)
      // 调用 controllerChannelManager.sendRequest(), 将请求放入对应的
      // messageQueue 等待发送
      controller.sendRequest(broker, ApiKeys.LEADER_AND_ISR, None,
        leaderAndIsrRequest, null)
  }
  leaderAndIsrRequestMap.clear() // 清空 leaderAndIsrRequestMap 集合
  // 第二步: 处理 updateMetadataRequestMap 集合, 逻辑同上 (略)
  // 第三步: 处理 stopReplicaRequestMap 集合, 逻辑同上 (略)
}

```

4.6.4 PartitionStateMachine

PartitionStateMachine 是 Controller Leader 用于维护分区状态的状态机。分区的状态是通过 PartitionState 接口定义的, 它有四个子类分别代表了分区四种可能的状态, 如表 4-2 所示。

表 4-2

状 态	含 义
NonExistentPartition	分区从来没有被创建或是分区被创建之后被又删除掉了, 这两种场景下的分区都处于此状态
NewPartition	分区被创建后就处于此状态。此时分区可能已经被分配了 AR 集合, 但是还没有指定 Leader 副本和 ISR 集合
OfflinePartition	已经成功选举出分区的 Leader 副本后, 但 Leader 副本发生宕机, 则分区转换为此状态。或者, 新创建的分区直接转换为此状态
OnlinePartition	分区成功选举出 Leader 副本之后, 分区会转换为此状态

分区各个 PartitionState 之间的转换如图 4-57 所示。

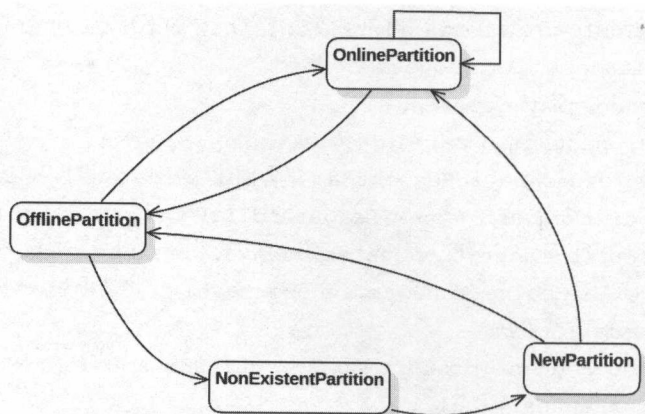


图 4-57

下面分析各个状态之间转换时，需要完成的相关操作。

- NonExistentPartition → NewPartition

从 ZooKeeper 中加载分区的 AR 集合到 ControllerContext 的 partitionReplicaAssignment 集合中。

- NewPartition → OnlinePartition

首先将 Leader 副本和 ISR 集合的信息写入到 ZooKeeper 中，这里会将分区的 AR 集合中第一个可用的副本选举为 Leader 副本，并将分区的所有可用副本作为 ISR 集合。之后，向所有可用的副本发送 LeaderAndIsrRequest，指导这些副本进行 Leader/Follower 的角色切换，并向所有可用的 Broker 发送 UpdateMetadataRequest 来更新其上的 MetadataCache。

- OnlinePartition/OfflinePartition → OnlinePartition

为分区选择新的 Leader 副本和 ISR 集合，并将结果写入 ZooKeeper。之后，向需要进行角色切换的副本发送 LeaderAndIsrRequest，指导这些副本进行 Leader/Follower 的角色切换，并向所有可用的 Broker 发送 UpdateMetadataRequest 来更新其上的 MetadataCache。

- NewPartition, OnlinePartition → OfflinePartition

只进行状态转换，并没有其他的操作。

- OfflinePartition → NonExistentPartition

只进行状态转换，并没有其他的操作。

PartitionStateMachine 中的各个字段含义和作用如下所述。

- `controllerContext`: `ControllerContext` 对象, 用于维护 `KafkaController` 的上下文信息。
- `zkUtils`: ZooKeeper 的客户端, 用于与 ZooKeeper 服务器交互。
- `partitionState`: `Map[TopicAndPartition, PartitionState]` 类型, 记录了每个分区对应的 `PartitionState` 状态。
- `brokerRequestBatch`: `ControllerBrokerRequestBatch` 对象, 用于向指定的 Broker 批量发送请求。

`noOpPartitionLeaderSelector`: 默认的 Leader 副本选举类器, 继承了 `PartitionLeaderSelector`。`NoOpLeaderSelector` 实现并没有真正进行 Leader 副本的选举, 其实现是返回当前的 Leader 副本、ISR 集合和 AR 集合。关于 `PartitionLeaderSelector` 的其他实现, 下文详述。

- `topicChangeListener`: ZooKeeper 的监听器, 用于监听 Topic 的变化。
- `deleteTopicsListener`: ZooKeeper 的监听器, 用于监听 Topic 的删除。
- `partitionModificationsListeners`: 用于监听分区的修改。

关于 ZooKeeper 监听器的相关介绍, 下文详述。

`PartitionStateMachine` 启动时会对 `partitionState` 集合进行初始化, 并调用 `triggerOnlinePartitionStateChange()` 方法将 `NewPartition` 和 `OfflinePartition` 状态的分区转换成 `OnlinePartition` 状态。

```
def startup() {
  initializePartitionState() // 初始化分区的状态
  hasStarted.set(true) // 标识 PartitionStateMachine 已启动
  triggerOnlinePartitionStateChange() // 尝试将分区切换到 OnlinePartition 状态
}
```

每个分区初始状态的依据是 `controllerContext.partitionLeadershipInfo` 中记录的 Leader 副本信息和 ISR 集合信息。

```
private def initializePartitionState() {
  // 遍历集群中所有分区
  for ((topicPartition, replicaAssignment) <-
    controllerContext.partitionReplicaAssignment) {
    controllerContext.partitionLeadershipInfo.get(topicPartition) match {
```

```

    case Some(currentLeaderIsrAndEpoch) => // 存在 Leader 副本和 ISR 集合的信息
        controllerContext.liveBrokerIds.contains(currentLeaderIsrAndEpoch
            .leaderAndIsr.leader) match {
            // Leader 副本所在的 Broker 可用, 初始化为 OnlinePartition 状态
            case true =>
                partitionState.put(topicPartition, OnlinePartition)
            // Leader 副本所在的 Broker 不可用, 初始化为 OfflinePartition 状态
            case false =>
                partitionState.put(topicPartition, OfflinePartition)
        }
    case None => // 没有 Leader 副本和 ISR 集合的信息
        partitionState.put(topicPartition, NewPartition)
}
}
}
}

```

`PartitionStateMachine.handleStateChange()` 方法是管理分区状态的核心方法, 该方法控制着 `PartitionState` 的转换。这里需要注意该方法的第三个参数, 它指定了用来选举 Leader 副本的 `PartitionLeaderSelector` 对象。

```

private def handleStateChange(topic: String, partition: Int,
    targetState: PartitionState, leaderSelector: PartitionLeaderSelector,
    callbacks: Callbacks) {
    val topicAndPartition = TopicAndPartition(topic, partition)
    // 检测当前 PartitionStateMachine 对象是否已启动, 只有 Controller Leader 的
    // PartitionStateMachine 对象才启动, 如果没有, 直接抛出异常退出 (略)

    // 从 partitionState 集合中获取分区的状态, 如果没有对应的状态, 则初始化为
    // NonExistentPartition
    val currState = partitionState.getOrElseUpdate(topicAndPartition,
        NonExistentPartition)
    try {
        // 在转换开始之前, 会根据 targetState 检查分区的前置状态是否合法 (略)
        targetState match {
            case NewPartition => // 将分区状态设置为 NewPartition
                partitionState.put(topicAndPartition, NewPartition)
            case OnlinePartition =>
                partitionState(topicAndPartition) match {

```

```

        case NewPartition => // 为分区初始化 Leader 副本和 ISR 集合
            initializeLeaderAndIsrForPartition(topicAndPartition)
        case OfflinePartition => // 为分区选举新的 Leader 副本
            electLeaderForPartition(topic, partition, leaderSelector)
        case OnlinePartition => // 为分区重新选举新的 Leader 副本
            electLeaderForPartition(topic, partition, leaderSelector)
    }
    // 将分区状态设置为 OnlinePartition
    partitionState.put(topicAndPartition, OnlinePartition)
    case OfflinePartition => // 将分区状态设置为 OfflinePartition
        partitionState.put(topicAndPartition, OfflinePartition)
    case NonExistentPartition => // 将分区状态设置为 NonExistentPartition
        partitionState.put(topicAndPartition, NonExistentPartition)
    }
} catch {
    ..... // 异常处理 (略)
}
}
}

```

PartitionState 由 NewPartition 切换为 OnlinePartition 时，调用了 initializeLeaderAndIsrForPartition() 方法，其操作的主要步骤是：

(1) 从 ControllerContext.partitionReplicaAssignment 集合中选择第一个可用的副本作为 Leader 副本，其余的副本构成 ISR 集合。

(2) 将 Leader 副本和 ISR 集合的信息写入到 ZooKeeper。

(3) 更新 ControllerContext.partitionLeadershipInfo 中缓存的 Leader 副本、ISR 集合等信息。

(4) 将上述步骤中确定的 Leader 副本、ISR 集合、AR 集合等信息添加到 ControllerBrokerRequestBatch，之后会封装成 LeaderAndIsrRequest 发送给相关的 Broker。

PartitionStateMachine.initializeLeaderAndIsrForPartition() 方法的具体实现如下：


```

private def initializeLeaderAndIsrForPartition(topicAndPartition: TopicAndPartition) {
    // 获取分区的 AR 集合信息
    val replicaAssignment = controllerContext
        .partitionReplicaAssignment(topicAndPartition)
    // 获取 AR 集合中的可用副本集合
    val liveAssignedReplicas = replicaAssignment.filter(
        r => controllerContext.liveBrokerIds.contains(r))
    liveAssignedReplicas.size match {
        case 0 => .....// 没有可用的副本, 抛出异常(略)
        case _ =>
            // 将可用的 AR 集合中的第一个副本选为 Leader
            val leader = liveAssignedReplicas.head
            // 创建 LeaderIsrAndControllerEpoch 对象, 其中的 ISR 集合是可用的 AR 集合,
            // leaderEpoch 和 zkVersion 都初始化为 0
            val leaderIsrAndControllerEpoch = new LeaderIsrAndControllerEpoch(
                new LeaderAndIsr(leader, liveAssignedReplicas.toList),
                controller.epoch)
            try {
                // 将 LeaderIsrAndControllerEpoch 中的信息转换成 JSON 格式保存到 ZooKeeper
                // 中, 具体路径是 "/brokers/topics/[topic_name]/partitions/[partitionId]/state"
                zkUtils.createPersistentPath(...)
                // 更新 partitionLeadershipInfo 中的记录
                controllerContext.partitionLeadershipInfo.put(topicAndPartition,
                    leaderIsrAndControllerEpoch)
                // 添加 LeaderAndISRRequest, 待发送
                brokerRequestBatch.addLeaderAndIsrRequestForBrokers(liveAssignedRe
                plicas,
                    topicAndPartition.topic, topicAndPartition.partition,
                    leaderIsrAndControllerEpoch, replicaAssignment)
            } catch {
                ..... // 异常处理(略)
            }
        }
    }
}

```

当 PartitionState 由 OfflinePartition 或 OnlinePartition 切换为 OnlinePartition 时调用了 electLeaderForPartition() 方法, 其操作的主要步骤是:

- (1) 使用指定的 PartitionLeaderSelector 为分区选举新的 Leader 副本。
- (2) 将 Leader 副本和 ISR 集合的信息写入到 Zookeeper。
- (3) 更新 ControllerContext.partitionLeadershipInfo 集合中缓存的 Leader 副本、ISR 集合等信息。
- (4) 将上述步骤中确定的 Leader 副本、ISR 集合、AR 集合等信息添加到 ControllerBrokerRequestBatch, 之后会封装成 LeaderAndIsrRequest 发送给相关的 Broker。

PartitionStateMachine.electLeaderForPartition() 方法的具体实现如下:

```
def electLeaderForPartition(topic: String, partition: Int,
                           leaderSelector: PartitionLeaderSelector) {
  val topicAndPartition = TopicAndPartition(topic, partition)
  try {
    var zookeeperPathUpdateSucceeded: Boolean = false
    var newLeaderAndIsr: LeaderAndIsr = null
    var replicasForThisPartition: Seq[Int] = Seq.empty[Int]

    while (!zookeeperPathUpdateSucceeded) {
      // 从 Zookeeper 中获取分区当前的 Leader 副本、ISR 集合、zkVersion 等信息, 如果
      // 不存在则抛出异常
      val currentLeaderIsrAndEpoch =
        getLeaderIsrAndEpochOrThrowException(topic, partition)
      val currentLeaderAndIsr = currentLeaderIsrAndEpoch.leaderAndIsr
      val controllerEpoch = currentLeaderIsrAndEpoch.controllerEpoch
      if (controllerEpoch > controller.epoch) {
        .....// 检测当前 controllerEpoch, 检测失败则抛出异常(略)
      }

      // 使用指定的 PartitionLeaderSelector 以及当前的 LeaderAndIsr 信息, 选举新
      // 的 Leader 副本和 ISR 集合。注意第二个返回值, 表示需要接收 LeaderAndIsrRequest
      // 的副本集合
      val (leaderAndIsr, replicas) = leaderSelector.
        selectLeader(topicAndPartition,
                    currentLeaderAndIsr)
      // 将新的 LeaderAndIsr 信息转换成 JSON 格式保存到 ZooKeeper 中, 具体路径
      // 是 "/brokers/topics/[topic_name]/partitions/[partitionId]/state"
      val (updateSucceeded, newVersion) = ReplicationUtils.
        updateLeaderAndIsr(
```

```

        zkUtils, topic, partition, leaderAndIsr, controller.epoch,
        currentLeaderAndIsr.zkVersion)

        newLeaderAndIsr = leaderAndIsr
        newLeaderAndIsr.zkVersion = newVersion
        zookeeperPathUpdateSucceeded = updateSucceeded
        replicasForThisPartition = replicas
    }
    val newLeaderIsrAndControllerEpoch = new LeaderIsrAndControllerEpoch(
        newLeaderAndIsr, controller.epoch)
    // 更新 partitionLeadershipInfo 中的记录
    controllerContext.partitionLeadershipInfo.put(TopicAndPartition(topic,
        partition), newLeaderIsrAndControllerEpoch)
    val replicas = controllerContext.partitionReplicaAssignment(
        TopicAndPartition(topic, partition))
    // 添加 LeaderAndISRRequest, 待发送
    brokerRequestBatch.addLeaderAndIsrRequestForBrokers(replicasForThisP
artition,
        topic, partition, newLeaderIsrAndControllerEpoch, replicas)
    }
} catch {
    .....// 异常处理(略)
}
}
}
}

```

在 `handleStateChange()` 方法中对于目标分区状态为 `NewPartition`、`OfflinePartition`、`NonExistentPartition` 的处理比较简单, 只是进行了状态切换, 并未进行其他处理。有的读者会说, 当 `PartitionState` 由 `NonExistentPartition` 转换为 `NewPartition` 时, 并没有从 `ZooKeeper` 中加载 `Partition` 的 AR 集合的相关操作。这是因为在调用 `handleStateChange()` 的方法中已经完成了此操作, 我们以创建 `Topic` 的过程为例。当创建 `Topic` 时触发 `TopicChangeListener` 这个监听器, 它会调用 `handleStateChange()` 完成 `PartitionState` 由 `NonExistentPartition` 到 `NewPartition` 的切换, 调用关系如图 4-58 所示。

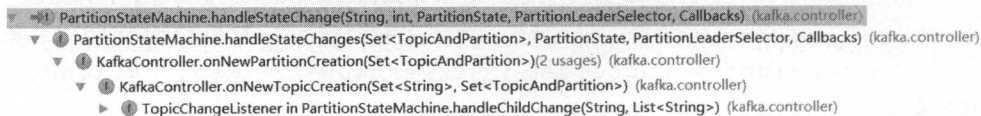


图 4-58

在 TopicChangeListener 中会完成从 ZooKeeper 加载 Partition 的 AR 集合的操作。

```
def handleChildChange(parentPath: String, children: java.util.List[String]) {
    .....
    // 从 zk 的 “/brokers/topics/[topic_name]” 路径下加载每个 Partition 的 AR 集合
    val addedPartitionReplicaAssignment = zkUtils
        .getReplicaAssignmentForTopics(newTopics.toSeq)
    controllerContext.partitionReplicaAssignment.++=(addedPartitionReplicaAssignment)
    ..... // 下面会调用 handleStateChange() 完成状态切换 (略)
}
```

handleStateChange() 方法是个 private 方法, 由 PartitionStateMachine.handleStateChanges() 方法和 triggerOnlinePartitionStateChange() 方法调用, 对外提供 PartitionState 切换。handleStateChanges() 方法对指定的分区集合循环调用 handleStateChange() 方法进行状态转换。

```
def handleStateChanges(partitions: Set[TopicAndPartition],
    targetState: PartitionState,
    leaderSelector: PartitionLeaderSelector =
    noOpPartitionLeaderSelector,
    callbacks: Callbacks = (new CallbackBuilder).build) {
    // 检查 ControllerBrokerRequestBatch 中的三个集合是否为空
    brokerRequestBatch.newBatch()
    partitions.foreach { topicAndPartition =>
        // 调用 handleStateChange() 方法对指定分区进行状态切换
        handleStateChange(topicAndPartition.topic, topicAndPartition.
            partition,
            targetState, leaderSelector, callbacks)
        brokerRequestBatch.sendRequestsToBrokers(controller.epoch) // 发送请求
    }
}
```

triggerOnlinePartitionStateChange() 方法对 partitionState 集合中的全部分区进行遍历, 将 OfflinePartition 和 NewPartition 状态的分区转换成 OnlinePartition 状态。状态切换成功的分区即可对外提供服务。


```

def triggerOnlinePartitionStateChange() {
  // 检查 ControllerBrokerRequestBatch 中的三个集合是否为空
  brokerRequestBatch.newBatch()
  for ((topicAndPartition, partitionState) <- partitionState
    if (!controller.deleteTopicManager
      .isTopicQueuedUpForDeletion(topicAndPartition.topic))) {
    if (partitionState.equals(OfflinePartition) ||
      partitionState.equals(NewPartition))
      // 调用 handleStateChange() 方法对指定分区进行状态切换
      handleStateChange(topicAndPartition.topic, topicAndPartition.
partition,
      OnlinePartition, controller.offlinePartitionSelector,
      (new CallbackBuilder).build) // 状态切换为 OnlinePartition
  }
  brokerRequestBatch.sendRequestsToBrokers(controller.epoch) // 发送请求
}

```

4.6.5 PartitionLeaderSelector

通过对前面的分析可知, PartitionMachine 将 Leader 副本选举、确定 ISR 集合的工作委托给了 PartitionLeaderSelector 接口实现, PartitionMachine 可以专注于管理分区状态。这是策略模式的一种典型的应用场景。

图 4-59 展示了 PartitionLeaderSelector 的实现类, 这五个不同的实现提供了不同的策略。PartitionLeaderSelector 接口的定义如下:

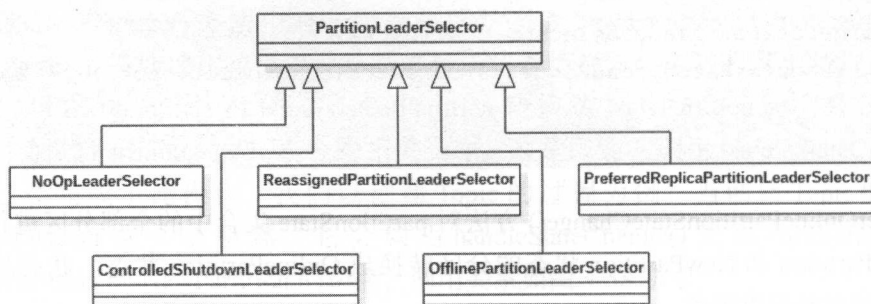


图 4-59

```
trait PartitionLeaderSelector {

  // 参数是：需要进行 Leader 副本选举的分区，以及其当前的 Leader 副本信息、ISR 信息
  // 返回值是选举后的新 Leader 副本和新 ISR 集合信息，以及需要接收 LeaderAndIsrRequest
  // 的 BrokerId
  def selectLeader(topicAndPartition: TopicAndPartition,
                   currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr,
Seq[Int])
}
```

NoOpLeaderSelector 是最简单的实现，它并没有进行 Leader 选举，而是将 currentLeaderAndIsr 直接返回，需要接收 LeaderAndIsrRequest 的 Broker 则是分区的 AR 集合。代码就不贴出来了。

OfflinePartitionLeaderSelector 会根据 currentLeaderAndIsr 选举新的 Leader 和 ISR 集合，策略如下：

(1) 如果在 ISR 集合中存在至少一个可用的副本，则从 ISR 集合中选择新的 Leader 副本，当前 ISR 集合为新 ISR 集合。

(2) 如果 ISR 集合中没有可用的副本且 “Unclean leader election” 配置被禁用，那么就抛出异常。

(3) 如果 “Unclean leader election” 被开启，则从 AR 集合中选择新的 Leader 副本和 ISR 集合。

(4) 如果 AR 集合中没有可用的副本，抛出异常。

OfflinePartitionLeaderSelector.selectLeader() 方法的具体实现如下：

```
def selectLeader(topicAndPartition: TopicAndPartition,
                 currentLeaderAndIsr: LeaderAndIsr): (LeaderAndIsr,
Seq[Int]) = {
  // 获取分区的 AR 集合
  controllerContext.partitionReplicaAssignment.get(topicAndPartition)
match {
    case Some(assignedReplicas) =>
      // 获取分区 AR 集合中可用的副本
      val liveAssignedReplicas = assignedReplicas
        .filter(r => controllerContext.liveBrokerIds.contains(r))
```

```

// 获取 ISR 集合中可用的副本
val liveBrokersInIsr = currentLeaderAndIsr.isr
    .filter(r => controllerContext.liveBrokerIds.contains(r))
// 当前 leaderEpoch
val currentLeaderEpoch = currentLeaderAndIsr.leaderEpoch
val currentLeaderIsrZkPathVersion = currentLeaderAndIsr.zkVersion

// 检测当前 ISR 集合中是否有可用副本
val newLeaderAndIsr = liveBrokersInIsr.isEmpty match {
    case true =>
        // 根据配置决定是否允许从 AR 集合中选择 Leader 副本, 如果不允许则抛异常 (略)
        .....
        liveAssignedReplicas.isEmpty match {
            // 检测当前 AR 集合中是否有可用副本
            case true => ... .. // AR 集合中没有可用副本, 直接抛出异常 (略)
            case false =>
                // 从可用 AR 集合中选取新的 Leader 副本, 新 ISR 集合中只有新 Leader 副本自己
                val newLeader = liveAssignedReplicas.head
                new LeaderAndIsr(newLeader, currentLeaderEpoch + 1,
                    List(newLeader), currentLeaderIsrZkPathVersion + 1)
        }
    case false =>
        // 从当前 ISR 集合中选取 Leader 副本以及 ISR 集合
        val newLeader = liveReplicasInIsr.head
        // 构造 LeaderAndIsr 对象并返回, 其中 leaderEpoch 和 zkVersion 都加一
        new LeaderAndIsr(newLeader, currentLeaderEpoch + 1,
            liveBrokersInIsr.toList, currentLeaderIsrZkPathVersion + 1)
}
// 需要向 AR 集合中所有可用的副本发送 LeaderAndIsrRequest
(newLeaderAndIsr, liveAssignedReplicas)
case None => ..... // AR 集合为空, 直接抛出异常 (略)
}
}

```

对于剩余的 PartitionLeaderSelector 实现, 这里只介绍其策略, 具体的实现代码留给读者自己分析。PreferredReplicaPartitionLeaderSelector 的策略是: 如果“优先副本”可用且在 ISR 集合中, 则选取其为 Leader 副本, 当前的 ISR 集合为新的 ISR 集合, 并向 AR 集合中所有可用副本发送 LeaderAndIsrRequest, 否则会抛出异常。

ReassignedPartitionLeaderSelector 涉及到副本的重新分配，副本重新分配的相关概念后面详细分析，这里先简单了解 ReassignedPartitionLeaderSelector 的策略：选取的新 Leader 副本必须在新指定的 AR 集合中且同时在当前 ISR 集合中，当前 ISR 集合为新 ISR 集合，接收 LeaderAndIsrRequest 的副本是新指定的 AR 集合中的副本。

ControlledShutdownLeaderSelector 的策略是：从当前 ISR 集合中排除正在关闭的副本后作为新的 ISR 集合，从新 ISR 集合中选择新的 Leader，需要向 AR 集合中可用的副本发送 LeaderAndIsrRequest。

4.6.6 ReplicaStateMachine

ReplicaStateMachine 是 Controller Leader 用于维护副本状态的状态机。副本状态由 ReplicaState 接口表示，它有七个子类，分别代表了副本的七种不同的状态，如表 4-3 所示。

表 4-3

状 态	含 义
NewReplica	创建新 Topic 或进行副本重新分配时，新创建的副本就处于这个状态。处于此状态的副本只能成为 Follower 副本
OnlineReplica	副本开始正常工作时处于此状态，处在此状态的副本可以成为 Leader 副本，也可以成为 Follower 副本
OfflineReplica	副本所在的 Broker 下线后，会转换为此状态
ReplicaDeletionStarted	刚开始删除副本时，会先将副本转换为此状态，然后开始删除操作
ReplicaDeletionSuccessful	副本被成功删除后，副本状态会处于此状态
ReplicaDeletionIneligible	如果副本删除操作失败，会将副本转换为此状态
NonExistentReplica	副本被成功删除后最终转换为此状态

ReplicaState 之间的转换如图 4-60 所示。下面介绍各个 ReplicaState 状态之间转换时需要完成的相关操作。

- NonExistentReplica → NewReplica

Controller 向此副本所在 Broker 发送 LeaderAndIsrRequest，并向集群中所有可用的 Broker 发送 UpdateMetadataRequest。

- NewReplica → OnlineReplica

Controller 将 NewReplica 加入到 AR 集合中。

- OnlineReplica, OfflineReplica → OnlineReplica

Controller 向此副本所在的 Broker 发送 LeaderAndIsrRequest，并向集群中所有可用的 Broker 发送 UpdateMetadataRequest。

- NewReplica, OnlineReplica, OfflineReplica, ReplicaDeletionIneligible → OfflineReplica

Controller 向副本所在 Broker 发送 StopReplicaRequest，之后会从 ISR 集合中清除此副本，最后向其他可用副本所在的 Broker 发送 LeaderAndIsrRequest，并向集群中所有可用的 Broker 发送 UpdateMetadataRequest。

- OfflineReplica → ReplicaDeletionStarted

Controller 向副本所在 Broker 发送 StopReplicaRequest。

- ReplicaDeletionStarted → ReplicaDeletionSuccessful

只做状态转换，并没有其他操作。

- ReplicaDeletionStarted → ReplicaDeletionIneligible

只做状态转换，并没有其他操作。

- ReplicaDeletionSuccessful → NonExistentReplica

Controller 从 AR 集合中删除此副本。

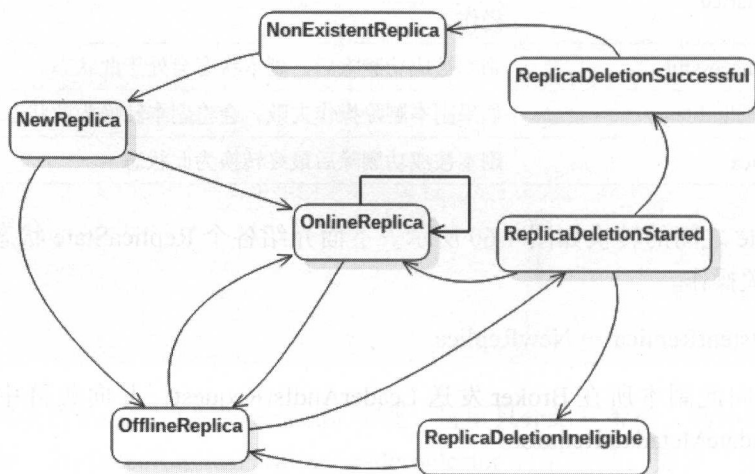


图 4-60

在 `ReplicaStateMachine` 中也有 `controllerContext`、`zkUtils`、`brokerRequestBatch` 字段，它们的功能与 `PartitionStateMachine` 中的同名字段相同，`ReplicaStateMachine` 剩余的字段如下所述。

- `replicaState`: `Map[PartitionAndReplica, ReplicaState]` 类型，记录每个副本对应的 `ReplicaState` 状态。
- `brokerChangeListener`: `ZooKeeper` 的监听器，用于监听 `Broker` 的变化，例如 `Broker` 宕机或重新上线等事件。关于 `ZooKeeper` 监听器的相关介绍，下文详述。

`ReplicaStateMachine` 启动时会对 `replicaState` 集合进行初始化，并调用 `handleStateChanges()` 方法尝试将可用副本转换为 `OnlineReplica` 状态。

```
def startup() {
  initializeReplicaState() // 初始化 replicaState 集合
  hasStarted.set(true)
  // 尝试将所有可用副本转换为 OnlineReplica 状态
  handleStateChanges(controllerContext.allLiveReplicas(), OnlineReplica)
}
```

设置每个副本的初始状态的依据是 `controllerContext.partitionLeadershipInfo` 中记录的 `Broker` 状态。`ReplicaStateMachine.initializeReplicaState()` 方法如下：

```
private def initializeReplicaState() {
  for ((topicPartition, assignedReplicas) <-
    controllerContext.partitionReplicaAssignment) {
    val topic = topicPartition.topic
    val partition = topicPartition.partition
    assignedReplicas.foreach { replicaId => // 遍历每个分区的 AR 集合
      val partitionAndReplica = PartitionAndReplica(topic, partition,
        replicaId)
      controllerContext.liveBrokerIds.contains(replicaId) match {
        // 将可用的副本初始化为 OnlineReplica 状态，不可用的副本初始化
        // 为 ReplicaDeletionIneligible 状态
        case true => replicaState.put(partitionAndReplica, OnlineReplica)
        case false => replicaState.put(partitionAndReplica, ReplicaDeletionIneligible)
      }
    }
  }
}
```

ReplicaStateMachine 的核心方法是 handleStateChange() 方法，其中控制着 ReplicaState 的转换。

```
def handleStateChange(partitionAndReplica: PartitionAndReplica,
                      targetState: ReplicaState, callbacks: Callbacks) {
    val topic = partitionAndReplica.topic
    val partition = partitionAndReplica.partition
    val replicaId = partitionAndReplica.replica
    val topicAndPartition = TopicAndPartition(topic, partition)
    ..... // 检测 ReplicaStateMachine 是否启动，如果未启动则抛出异常（略）
    try {
        val replicaAssignment = controllerContext
            .partitionReplicaAssignment(topicAndPartition) // 获取分区的 AR 集合
        targetState match {
            // 在转换开始之前，都会根据 targetState 检查前置状态是否合法（略）
            case NewReplica =>
                // 从 ZK 中获取该分区的 Leader 副本、ISR 等信息
                val leaderIsrAndControllerEpochOpt = ReplicationUtils
                    .getLeaderIsrAndEpochForPartition(zkUtils, topic, partition)
                leaderIsrAndControllerEpochOpt match {
                    case Some(leaderIsrAndControllerEpoch) =>
                        ..... // 处于 NewReplica 状态的副本不能是 Leader，若为 Leader 则报错（略）
                        // 向该副本发送 LeaderAndIsrRequest，并发送 UpdateMetadataRequest 给所
                        // 有可用的 Broker
                        brokerRequestBatch.addLeaderAndIsrRequestForBrokers (List
(replicaId),
                            topic, partition, leaderIsrAndControllerEpoch,
                            replicaAssignment)
                        case None => {
                            .....
                        }
                    }
                // 更新副本状态为 NewReplica
                replicaState.put(partitionAndReplica, NewReplica)

            case ReplicaDeletionStarted =>
                // 更新副本状态为 ReplicaDeletionStarted

                replicaState.put(partitionAndReplica, ReplicaDeletionStarted)
        }
    }
}
```

```

// 向该副本发送 StopReplicaRequest, 注意这里设置了回调函数
brokerRequestBatch.addStopReplicaRequestForBrokers(List(replica
Id), topic,
    partition, deletePartition = true, callbacks.
stopReplicaResponseCallback)

case ReplicaDeletionIneligible =>
    // 更新副本状态为 ReplicaDeletionIneligible
    replicaState.put(partitionAndReplica, ReplicaDeletionIneligible)

case ReplicaDeletionSuccessful =>
    // 更新副本状态为 ReplicaDeletionSuccessful
    replicaState.put(partitionAndReplica, ReplicaDeletionSuccessful)

case NonExistentReplica =>
    // 从 AR 集合中删除该副本
    val currentAssignedReplicas =
        controllerContext.partitionReplicaAssignment(topicAndPartition)
        controllerContext.partitionReplicaAssignment.
put(topicAndPartition,
    currentAssignedReplicas.filterNot(_ == replicaId))
    replicaState.remove(partitionAndReplica) // 删除副本状态

case OnlineReplica =>
    replicaState(partitionAndReplica) match {
        case NewReplica =>
            // 将该副本添加到 AR 集合中
            val currentAssignedReplicas =
                controllerContext.partitionReplicaAssignment(topicAndPartition)
            if (!currentAssignedReplicas.contains(replicaId))
                controllerContext.partitionReplicaAssignment.put(topicAndPar
tition, currentAssignedReplicas :+ replicaId)
        case _ =>
            // 检测是否存在 Leader 副本
            controllerContext.partitionLeadershipInfo.get(topicAndPartition)
match {

```



```

        case Some(leaderIsrAndControllerEpoch) =>
            // 如果存在 Leader 副本信息, 向该副本发送 LeaderAndIsrRequest,
            // 并发送 UpdateMetadataRequest 给所有可用的 Broker
            brokerRequestBatch.addLeaderAndIsrRequestForBrokers(List(r
replicaId),
                                                                    topic, partition, leaderIsrAndControllerEpoch,
replicaAssignment)
            replicaState.put(partitionAndReplica, OnlineReplica)
        case None =>
            }
        }
        // 更新副本状态为 OnlineReplica
        replicaState.put(partitionAndReplica, OnlineReplica)

    case OfflineReplica =>
        // 向该副本发送 StopReplicaRequest, 注意, 这里不会删除副本
        brokerRequestBatch.addStopReplicaRequestForBrokers(List(replicaId),
            topic, partition, deletePartition = false)
        val leaderAndIsrIsEmpty: Boolean =
            controllerContext.partitionLeadershipInfo.get(topicAndPartition)
match {
    case Some(currLeaderIsrAndControllerEpoch) =>
        // 将该副本从 ISR 集合中移除
        controller.removeReplicaFromIsr(topic, partition, replicaId)
match {
            case Some(updatedLeaderIsrAndControllerEpoch) =>
                val currentAssignedReplicas = controllerContext
                    .partitionReplicaAssignment(topicAndPartition)
                if (!controller.deleteTopicManager
                    .isPartitionToBeDeleted(topicAndPartition)) {
                    // 向其他可用副本发送 LeaderAndIsrRequest, 并向集群中所有可用
                    // 的 Broker 发送 UpdateMetadataRequest
                    brokerRequestBatch.addLeaderAndIsrRequestForBrokers(
                        currentAssignedReplicas.filterNot(_ == replicaId),
topic, partition,
                        updatedLeaderIsrAndControllerEpoch, replicaAssignment)
                }
                // 更新副本状态为 OfflineReplica

```

```

        replicaState.put(partitionAndReplica, OfflineReplica)
        false
        case None => true
    }
    case None => true
}
..... // 边界检测(略)
}
}
catch {
    ..... // 异常处理(略)
}
}

```

`ReplicaStateMachine.handleStateChanges()` 方法对指定的副本集合循环调用 `handleStateChange()` 方法来完成状态转换, 与 `PartitionStateMachine` 中的实现类似, 代码不贴出来了。

4.6.7 ZooKeeper Listener

`IOIttec-zkClient` 是一款 ZooKeeper 客户端工具, 它并没有像 Apache Curator 那样实现高级的功能, 而是提供了简单易用的 API 来实现一些常见的功能。`IOIttec-zkClient` 像大多数 ZooKeeper 客户端框架一样实现了断线重连。它还提供了方便使用的监听器, 避免了手动反复注册 `Watcher` 的烦琐操作, 下文会介绍 `KafkaController` 中的多个 `IOIttec-zkClient` `Listener` 实现。`IOIttec` 还对 ZooKeeper 的异常和序列化做了简单封装。此处不对 `IOIttec-zkClient` 的使用展开详述, 感兴趣的读者可以参考相关文档进行学习。

Listener 接口介绍

`KafkaController` 会通过 ZooKeeper 监控整个 Kafka 集群的运行状态, 响应管理员指定的相关操作。具体的实现方式是在 ZooKeeper 的指定节点上添加 `Listener`, 监听此节点中的数据变化或是其子节点的变化, 从而触发相应的业务逻辑。

`Listener` 按照接口的类型可以分为三类, 如表 4-4 所示。

表 4-4

Listener 类型	作 用
IZkDataListener	监听指定节点的数据变化
IZkChildListener	监听指定节点的子节点变化
IZkStateListener	监听 ZooKeeper 连接状态的变化

IZkDataListener 接口的定义如下：

```
public interface IZkDataListener {
    // 当监听节点的中保存的数据发生变化时，handleDataChange() 方法会被触发
    public void handleDataChange(String dataPath, Object data) throws
Exception;

    // 当监听节点被删除时，handleDataDeleted() 方法会被触发
    public void handleDataDeleted(String dataPath) throws Exception;
}
```

IZkChildListener 接口中的定义如下：

```
public interface IZkChildListener {
    // 当监听节点的子节点发生变化时，handleChildChange() 方法会被触发
    // 第一个参数是监听的路径，第二个参数是当前子节点集合
    public void handleChildChange(String parentPath, List<String>
currentChilds)
        throws Exception;
}
```

IZkStateListener 接口的定义如下：

```
public interface IZkStateListener {

    // 当客户端与 ZooKeeper 连接状态发生变化时，handleStateChanged() 方法会被触发
    public void handleStateChanged(KeeperState state) throws Exception;

    // 当客户端的 Session 过期后重新建立新的 Session 时，handleNewSession() 方法会被触发
    public void handleNewSession() throws Exception;
    // 当 Session 重建失败时，handleSessionEstablishmentError() 方法会被触发
    public void handleSessionEstablishmentError(final Throwable error)
        throws Exception;
}
```

Kafka 中提供了五个 IZkDataListener 接口的实现，它们分别是：LeaderChangeListener、PartitionModificationsListener、PreferredReplicaElectionListener、PartitionsReassignedListener、ReassignedPartitionsIsrChangeListener。如图 4-61 所示。

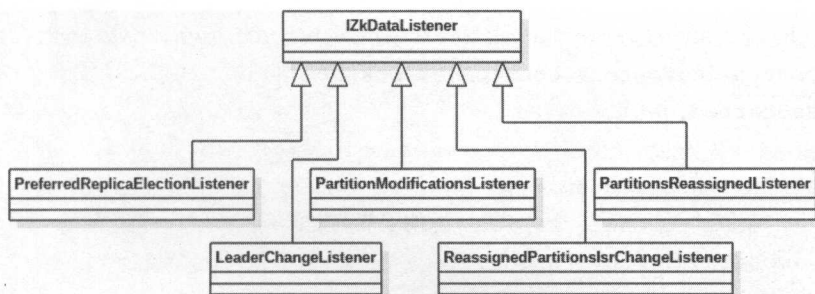


图 4-61

Kafka 中提供了四个 IZkChildListener 接口的实现，它们分别是：DeleteTopicsListener、TopicChangeListener、IsrChangeNotificationListener、BrokerChangeListener，如图 4-62 所示。

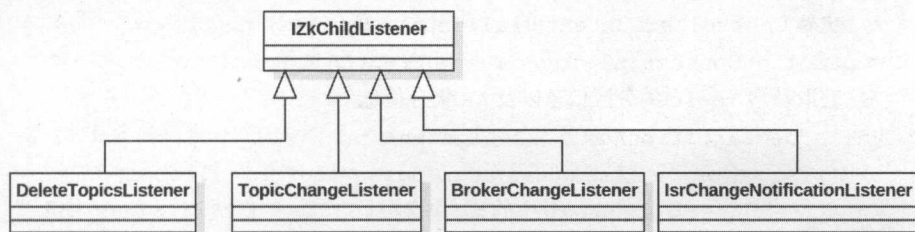


图 4-62

Kafka 中只有了一个 IZkStateListener 接口的实现，如图 4-63 所示。

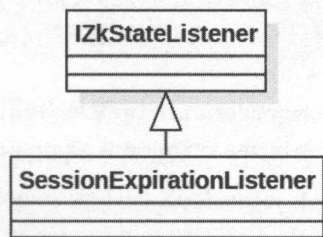


图 4-63

TopicChangeListener

TopicChangeListener 负责管理 Topic 的增删，它监听“/brokers/topics”节点的子节点变化。其具体逻辑如下：

```
def handleChildChange(parentPath: String, children: java.util.List[String]) {
  inLock(controllerContext.controllerLock) {
    if (hasStarted.get) {
      try {
        val currentChildren = {
          // 获取“/brokers/topics”下的子节点集合
          import JavaConversions._
          (children: Buffer[String]).toSet
        }
        // 过滤，得到新添加 Topic
        val newTopics = currentChildren -- controllerContext.allTopics
        // 过滤，得到删除的 Topic
        val deletedTopics = controllerContext.allTopics -- currentChildren
        // 更新 ControllerContext 的 allTopics 集合
        controllerContext.allTopics = currentChildren
        // 获取新增 Topic 的分区信息以及 AR 集合信息
        val addedPartitionReplicaAssignment =
          zkUtils.getReplicaAssignmentForTopics(newTopics.toSeq)
        // 更新 ControllerContext 中的 AR 集合记录
        controllerContext.partitionReplicaAssignment = controllerContext
          .partitionReplicaAssignment.filter(
            p => ! deletedTopics.contains(p._1.topic))
        controllerContext.partitionReplicaAssignment.++=
          (addedPartitionReplicaAssignment)
        // 调用 onNewTopicCreation() 方法处理新增 Topic
        if (newTopics.size > 0)
          controller.onNewTopicCreation(newTopics,
            addedPartitionReplicaAssignment.keySet().toSet)
      } catch {
        // 异常处理（略）
      }
    }
  }
}
```

在 `TopicChangeListener.onNewTopicCreation()` 方法中还会为每个新增的 Topic 注册一个 `PartitionModificationsListener`，然后调用 `onNewPartitionCreation()` 方法完成新增 Topic 的分区状态以及副本状态转换。

```
def onNewTopicCreation(topics: Set[String], newPartitions: Set[TopicAndPartition]) {
  // 为每一个新增的 Topic 都注册一个 PartitionModificationsListener
  topics.foreach {
    topic => partitionStateMachine.registerPartitionChangeListener(topic)

    onNewPartitionCreation(newPartitions)
  }
}

// 下面是 onNewPartitionCreation() 方法的实现
def onNewPartitionCreation(newPartitions: Set[TopicAndPartition]) {
  // 将所有指定的新增分区转换为 NewPartition 状态
  partitionStateMachine.handleStateChanges(newPartitions, NewPartition)
  // 将指定分区的所有副本都转换为 NewReplica 状态
  replicaStateMachine.handleStateChanges(
    controllerContext.replicasForPartition(newPartitions), NewReplica)

  // 将所有指定的新增分区转换为 OnlinePartition 状态
  // 注意，这里虽然指定了 offlinePartitionSelector
  // 但是从 NewPartition 转换为 OnlinePartition 并没有用到此 PartitionSelector
  partitionStateMachine.handleStateChanges(
    newPartitions, OnlinePartition, offlinePartitionSelector)
  // 将指定分区的所有副本都转换为 NewReplica 状态
  replicaStateMachine.handleStateChanges(
    controllerContext.replicasForPartition(newPartitions), OnlineReplica)
}
```

为了便于读者理解，这里举例解释 `TopicChangeListener` 的功能。现在假设有三个 Broker，管理人员通过 `kafka-topics` 脚本添加了一个名为“test”的 Topic，它有三个分区，每个分区有三个副本。`kafka-topics` 脚本向 ZooKeeper 的“/brokers/topics/test”节点写入的信息是“partitions”: {“0”: [0,1, 2],“1”: [1,2,0],“2”: [2,1,0]}，此时触发 `TopicChangeListener`。`TopicChangeListener` 将 test 中每个 Partition 的 AR 集合加载到 `ControllerContext` 中，在进行第一次分区状态转换（`NoExistentPartition` → `NewPartition`）和第一次副本状态切换（`NoExistentReplica` → `NewReplica`）时，只做了状态切换并没有发送任何请求。进行第二

次分区状态转换（NewPartition → OnlinePartition）时会选取 Leader 副本和 ISR 集合信息，结果为如表 4-5 所示。

表 4-5

分 区	0	1	2
Leader 副本	0	1	2
ISR 集合	(0,1,2)	(1,2,0)	(2,1,0)

之后会将此结果写入 ZooKeeper，向所有可用 Broker 发送 LeaderAndIsrRequest 来指导副本的角色切换，然后向所有可用 Broker 发送 UpdateMetadataRequest 来更新其 MetadataCache。第二次副本状态切换（NewReplica → OnlineReplica）时，副本已在 AR 集合中，所以并未做任何操作。

TopicDeletionManager 与 DeleteTopicsListener

在开始介绍删除 Topic 的实现之前，先来了解一下 TopicDeletionManager 的功能和实现。在 TopicDeletionManager 中维护了多个集合，用于管理待删除的 Topic 和不可删除的集合，它会启动一个 DeleteTopicsThread 线程来执行删除 Topic 的具体逻辑。

当 Topic 满足下列三种情况之一时不能被删除：

- （1）如果 Topic 中的任一分区正在重新分配副本，则此 Topic 不能被删除。
- （2）如果 Topic 中的任一分区正在进行“优先副本”选举，则此 Topic 不能被删除。
- （3）如果 Topic 中的任一分区的任一副本所在的 Broker 宕机，则此 Topic 不能被删除。

TopicDeletionManager 中各个字段的含义和功能如下所示。

- partitionStateMachine：用于管理分区状态的状态机。
- replicaStateMachine：用于管理副本状态的状态机。
- topicsToBeDeleted：Set[String] 类型，用于记录将要被删除的 Topic 集合，由 TopicDeletionManager 的构造器参数 initialTopicsToBeDeleted 指定其初始化值。
- partitionsToBeDeleted：Set[TopicAndPartition] 类型，用于记录将要被删除的分区集合。
- topicsIneligibleForDeletion：Set[String] 类型，用于记录不可删除的 Topic 集合，由 TopicDeletionManager 的构造器参数 initialTopicsIneligibleForDeletion 指定其初始化值。
- deleteTopicStateChanged：AtomicBoolean 类型，用于标识 Topic 删除操作是否开始。

- deleteTopicsThread: DeleteTopicsThread 类型, 用于删除 Topic 的后台线程。
- isDeleteTopicEnabled: 配置项 delete.topic.enable 的值, 用于指定是否支持删除 Topic。
- deleteTopicsCond: Condition 对象, 用于其他线程与 deleteTopicsThread 线程同步。

在 TopicDeletionManager 启动时, 会调用 start() 方法进行初始化。它会根据 isDeleteTopicEnabled 字段决定是否启动 DeleteTopicsThread 线程, 如果此时 topicsToBeDeleted 集合不为空, 则 DeleteTopicsThread 可以开始进行 Topic 删除的相关操作, 并将 deleteTopicStateChanged 字段设置为 true。

DeleteTopicsListener 被触发后通过 enqueueTopicsForDeletion() 将待删除的 Topic 放入 topicsToBeDeleted 集合, 将待删除的 Topic 的分区集合放入 partitionsToBeDeleted 集合, 并唤醒 DeleteTopicsThread 处理。

```
def enqueueTopicsForDeletion(topics: Set[String]) {
  if (isDeleteTopicEnabled) { // 检测 isDeleteTopicEnabled
    // 待删除的 Topic 放入 topicsToBeDeleted 集合
    topicsToBeDeleted += topics
    // 待删除的 Topic 的分区集合放入 partitionsToBeDeleted 集合
    partitionsToBeDeleted += topics.flatMap(controllerContext.
      partitionsForTopic)
    resumeTopicDeletionThread() // 唤醒 DeleteTopicsThread 线程
  }
}
```

DeleteTopicsThread 是真正执行 Topic 删除操作的线程, 它继承了 ShutdownableThread, 入口方法是 doWork() 方法。删除 Topic 的步骤如下:

(1) 获取待删除 Topic 的分区集合, 构成 UpdateMetadataRequest 发送给所有的 Broker, 将 Broker 中 MetadataCache 的相关信息删除。这些分区不再对外提供服务。

(2) 调用 onPartitionDeletion() 方法开始对指定分区进行删除。

a) 将不可用副本转换成 ReplicaDeletionIneligible 状态。

b) 将可用副本转换成 OfflineReplica 状态。此步骤会发送 StopReplicaRequest 到待删除的副本 (不会删除副本), 同时还会向可用的 Broker 发送 LeaderAndIsrRequest 和 UpdateMetadataRequest, 将副本从 ISR 集合中删除。

c) 将可用副本由 `OfflineReplica` 转换成 `ReplicaDeletionStarted`。此步骤会向可用副本发送 `StopReplicaRequest` (删除副本)。注意, 这里设置了回调函数处理 `StopReplicaResponse`。

(3) 调用 `deleteTopicStopReplicaCallback()` 回调函数处理 `StopReplicaResponse`。

a) 如果 `StopReplicaResponse` 中的错误码表示出现异常, 则将副本状态转换为 `ReplicaDeletionIneligible`, 并标记此副本所在 Topic 不可删除, 也就是将 Topic 添加到 `topicsIneligibleForDeletion` 队列, 最后唤醒 `DeleteTopicsThread` 线程。

b) 如果 `StopReplicaResponse` 正常, 则将副本状态转换为 `ReplicaDeletionSuccessful`, 并唤醒 `DeleteTopicsThread` 线程。

(4) 经过上述三个步骤后, 开始第二次 `doWork()` 调用。如果待删除的 Topic 的所有副本已经处于 `ReplicaDeletionSuccessful` 状态, 调用 `completeDeleteTopic()` 方法完成 Topic 的删除。

a) 取消 `partitionModificationsListeners` 监听。

b) 将此 Topic 的所有副本从 `ReplicaDeletionSuccessful` 转换为 `NonExistentReplica`。此步骤会将副本对应的 `Replica` 对象从 `ControllerContext` 中删除。

c) 将 Topic 的所有分区转换为 `OfflinePartition` 状态, 紧接着会再转换为 `NonExistentPartition`。

d) 将 Topic 和相关的分区从 `topicsToBeDeleted` 集合和 `partitionsToBeDeleted` 集合中删除。

e) 删除 ZooKeeper 以及 `ControllerContext` 中与此 Topic 相关的全部信息。

(5) 如果还有副本处于 `ReplicaDeletionStarted` 状态, 则表示还没有收到 `StopReplicaResponse`, 则继续等待。

(6) 如果 Topic 的任一副本处于 `ReplicaDeletionIneligible` 状态, 则表示此 Topic 不能被删除, 调用 `markTopicForDeletionRetry()` 将处于 `ReplicaDeletionIneligible` 状态的副本重新转换成 `OfflineReplica` 状态。此步骤的相关操作在步骤 (2) → b 中已经详细描述, 这里不再赘述。

`DeleteTopicsThread.doWork()` 方法的具体实现如下:

```

override def doWork() {
  awaitTopicDeletionNotification() // 等待被唤醒
  .....// 边界检查, 加锁操作 (略)
  val topicsQueuedForDeletion = Set.empty[String] ++ topicsToBeDeleted
  topicsQueuedForDeletion.foreach { topic =>
    // 步骤 4: 检测所有副本是否处于 ReplicaDeletionSuccessful
    if (controller.replicaStateMachine.areAllReplicasForTopicDeleted(topic)) {
      // 清空所有副本和 Partition 的状态, 清空 ControllerContext
      // 和 ZooKeeper 中与此 Topic 相关信息
      completeDeleteTopic(topic)
    } else {
      // 步骤 5: 如果任意一个副本处于 ReplicaDeletionStarted, 则继续等待
      if (controller.replicaStateMachine
        .isAtLeastOneReplicaInDeletionStartedState(topic)) {
        ..... // 日志输出操作 (略)
      } else {
        // 步骤 6: 任一副本处于 ReplicaDeletionIneligible, 则重置为 OfflineReplica 后重试
        if (controller.replicaStateMachine
          .isAnyReplicaInState(topic, ReplicaDeletionIneligible)) {
          markTopicForDeletionRetry(topic)
        }
      }
    }
  }
  if (isTopicEligibleForDeletion(topic)) {
    // 检测当前 Topic 是否可以删除
    onTopicDeletion(Set(topic)) // 开始 Topic 的删除操作, 对应步骤 1~3
  } else if (isTopicIneligibleForDeletion(topic)) {
    ..... // 日志输出操作 (略)
  }
}
}
}

```

`isTopicEligibleForDeletion()` 方法根据下面三个条件判断 Topic 能否开始删除操作。

```
private def isTopicEligibleForDeletion(topic: String): Boolean = {
  topicsToBeDeleted.contains(topic) // 当前 Topic 没有完全完成删除操作
  && (!isTopicDeletionInProgress(topic) // Topic 删除操作还没有开始
    && !isTopicIneligibleForDeletion(topic)) // Topic 没有被标记为不可删除
}
```

`onTopicDeletion()` 方法的核心是向所有可用的 Broker 发送 `UpdateMetadataRequest`, 注意其 `leader` 字段为 `LeaderDuringDelete`, 通知它们指定的 Topic 要被删除, 并删除 `MetadataCache` 中与此 Topic 相关的缓存信息。

```
private def onTopicDeletion(topics: Set[String]) {
  val partitions = topics.flatMap(controllerContext.partitionsForTopic)
  // 向可用 Broker 发送 UpdateMetadataRequest
  controller.sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.
    toSeq, partitions)
  // 按照 Topic 进行分组
  val partitionReplicaAssignmentByTopic =
    controllerContext.partitionReplicaAssignment.groupBy(p => p._1.topic)
  // 开始分区的删除操作
  topics.foreach { topic =>
    onPartitionDeletion(partitionReplicaAssignmentByTopic(topic).
      map(_._1).toSet)
  }
}
```

`onPartitionDeletion()` 方法直接调用了 `startReplicaDeletion()` 方法, 在 `startReplicaDeletion()` 方法中开始对副本进行删除。

```
private def onPartitionDeletion(partitionsToBeDeleted: Set[TopicAndPartition]) {
  // 获取待删除副本的 AR 集合
  val replicasPerPartition = controllerContext
    .replicasForPartition(partitionsToBeDeleted)
  startReplicaDeletion(replicasPerPartition)
}
// 下面是 startReplicaDeletion() 方法的实现
private def startReplicaDeletion(
```

```

        replicasForTopicsToBeDeleted: Set[PartitionAndReplica]) {
    replicasForTopicsToBeDeleted.groupBy(_.topic).foreach { case (topic,
replicas) =>
    // 获取 Topic 中所有的可用副本
    var aliveReplicasForTopic = controllerContext.allLiveReplicas()
        .filter(p => p.topic.equals(topic))
    // 获取 Topic 中所有的不可用副本
    val deadReplicasForTopic = replicasForTopicsToBeDeleted --
aliveReplicasForTopic
    // 获取 Topic 中所有已完成删除的副本
    val successfullyDeletedReplicas = controller.replicaStateMachine
        .replicasInState(topic, ReplicaDeletionSuccessful)
    // 第一次删除（或重试）的副本集合
    val replicasForDeletionRetry = aliveReplicasForTopic --
successfullyDeletedReplicas

    // 步骤 2.a: 将不可用副本转换为 ReplicaDeletionIneligible 状态
    replicaStateMachine.handleStateChanges(deadReplicasForTopic,
        ReplicaDeletionIneligible)
    // 步骤 2.b: 将待删除的副本转换为 OfflineReplica 状态
    replicaStateMachine.handleStateChanges(replicasForDeletionRetry,
OfflineReplica)
    // 步骤 2.c: 将待删除的副本转换为 ReplicaDeletionStarted 状态
    controller.replicaStateMachine.handleStateChanges(replicasForDeletionR
etry,
        ReplicaDeletionStarted, new Callbacks.CallbackBuilder()
            .stopReplicaCallback(deleteTopicStopReplicaCallback).build)
    if (deadReplicasForTopic.size > 0) {
        // 标记当前 Topic 不可删除，即添加到 topicsIneligibleForDeletion 集合汇总
        markTopicIneligibleForDeletion(Set(topic))
    }
}
}
}

```

`deleteTopicStopReplicaCallback()` 回调函数中会调用 `failReplicaDeletion()` 方法处理异常副本，调用 `completeReplicaDeletion()` 方法处理返回正常 `StopReplicaResponse` 的副本。


```
// 下面是 failReplicaDeletion() 方法的实现
def failReplicaDeletion(replicas: Set[PartitionAndReplica]) {
  val replicasThatFailedToDelete = replicas.filter(...)
  if (replicasThatFailedToDelete.size > 0) {
    val topics = replicasThatFailedToDelete.map(_.topic)
    .....
    // 步骤 3.a: 将异常副本转换为 ReplicaDeletionIneligible 状态
    controller.replicaStateMachine.handleStateChanges(
      replicasThatFailedToDelete, ReplicaDeletionIneligible)
    markTopicIneligibleForDeletion(topics) // 将当前 Topic 标记为不可用
    resumeTopicDeletionThread() // 唤醒 DeleteTopicsThread 线程
  }
}

// 下面是 completeReplicaDeletion() 方法的实现
private def completeReplicaDeletion(replicas: Set[PartitionAndReplica]) {
  val successfullyDeletedReplicas = replicas.filter(
    r => isTopicQueuedUpForDeletion(r.topic))

  // 将成功删除的副本转换为 ReplicaDeletionSuccessful 状态
  controller.replicaStateMachine.handleStateChanges(successfullyDeletedReplicas,
    ReplicaDeletionSuccessful)
  resumeTopicDeletionThread() // 唤醒 DeleteTopicsThread 线程
}
```

回到 doWork() 方法继续分析，在步骤 4 中调用 completeDeleteTopic() 方法对成功删除的 Topic 进行处理。

```
private def completeDeleteTopic(topic: String) {
  // 步骤 4.a: 取消 partitionModificationsListeners 监听
  partitionStateMachine.deregisterPartitionChangeListener(topic)
  // 步骤 4.b: 将 ReplicaStateMachine 中维护的此 Topic 所有副本的状态清除
  val replicasForDeletedTopic = controller.replicaStateMachine
    .replicasInState(topic, ReplicaDeletionSuccessful)
  replicaStateMachine.handleStateChanges(replicasForDeletedTopic,
    NonExistentReplica)
  // 步骤 4.c: 将此 Topic 所有分区状态转换为 NonExistentPartition
}
```

```

val partitionsForDeletedTopic = controllerContext.partitionsForTopic(topic)
partitionStateMachine.handleStateChanges(partitionsForDeletedTopic,
    OfflinePartition)
partitionStateMachine.handleStateChanges(partitionsForDeletedTopic,
    NonExistentPartition)
// 步骤4.d: 清空 topicsToBeDeleted 和 partitionsToBeDeleted 中的相关内容
topicsToBeDeleted -= topic
partitionsToBeDeleted.retain(_.topic != topic)
// 步骤4.e: 清空 ZooKeeper 中此 Topic 对应的所有数据
val zkUtils = controllerContext.zkUtils
zkUtils.zkClient.deleteRecursive(getTopicPath(topic))
zkUtils.zkClient.deleteRecursive(getEntityConfigPath(ConfigType.Topic,
    topic))
zkUtils.zkClient.delete(getDeleteTopicPath(topic))
controllerContext.removeTopic(topic) // 清空 controllerContext 中的相应缓存
}

```

在步骤6中调用 `markTopicForDeletionRetry()` 方法处理不可删除的 Topic，它会将处于 `ReplicaDeletionIneligible` 状态的副本重新转换成 `OfflineReplica` 状态。

```

private def markTopicForDeletionRetry(topic: String) {
    // 找到 ReplicaDeletionIneligible 状态的副本，并将其转换为 OfflineReplica
    val failedReplicas = controller.replicaStateMachine
        .replicasInState(topic, ReplicaDeletionIneligible)

    controller.replicaStateMachine.handleStateChanges(failedReplicas,
        OfflineReplica)
}

```

在前面我们介绍了三种 Topic 不可删除的情况，在 `DeleteTopicsThread` 线程的执行过程中也有涉及。当 Topic 不再满足这三种情况时会通过 `resumeDeletionForTopics()` 方法从 `topicsIneligibleForDeletion` 集合中将其移除，并唤醒 `DeleteTopicsThread` 线程进行上述删除操作。图 4-64 是其被调用的位置，依次对应本节开始描述的三种情况。

```

def resumeDeletionForTopics(topics: Set[String] = Set.empty) {
  if (isDeleteTopicEnabled) {
    // 找出给定 Topic 集合与待删除 Topic 集合的交集, 如果这个交集不为空, 将交集中的
    // Topic 从 topicsIneligibleForDeletion 集合中删除
    val topicsToResumeDeletion = topics & topicsToBeDeleted
    if (topicsToResumeDeletion.size > 0) {
      topicsIneligibleForDeletion --= topicsToResumeDeletion
      resumeTopicDeletionThread() // 唤醒 DeleteTopicsThread 线程处理待删除的
Topic
    }
  }
}

```

- ▼ ① TopicDeletionManager.resumeDeletionForTopics(Set<String>) (kafka.controller)
- ① ▶ ① KafkaController.onPartitionReassignment(TopicAndPartition, ReassignedPartitionsContext) (kafka.controller)
- ② ▶ ① KafkaController.onPreferredReplicaElection(Set<TopicAndPartition>, boolean) (kafka.controller)
- ③ ▶ ① KafkaController.onBrokerStartup(Seq<Object>) (kafka.controller)

图 4-64

介绍完 TopicDeletionManager 的相关实现后再来对 DeleteTopicsListener 进行分析。DeleteTopicsListener 会监听 ZooKeeper 中 “/admin/delete_topics” 节点下的子节点变化, 当 TopicCommand 在该路径下添加需要被删除的 Topic 时, DeleteTopicsListener 会被触发, 它会将该待删除的 Topic 交由 TopicDeletionManager 执行 Topic 删除操作。下面是 DeleteTopicsListener.handleChildChange() 方法的具体实现:

```

def handleChildChange(parentPath: String, children: java.util.List[String])
{
  inLock(controllerContext.controllerLock) {
    var topicsToBeDeleted = {
      ... .. // 从 ZooKeeper 中获取待删除 Topic 集合
    }
    val nonExistentTopics = topicsToBeDeleted.filter(
      // 检查待删除 Topic 是否存在
      t => !controllerContext.allTopics.contains(t))
    if (nonExistentTopics.size > 0) {
      // 对于不存在的 Topic, 直接将其在 “/admin/delete_topics” 下对应的节点删除
    }
  }
}

```



```

        nonExistentTopics.foreach(topic =>
            zkUtils.deletePathRecursive(getDeleteTopicPath(topic)))
    }

    topicsToBeDeleted --= nonExistentTopics // 过滤掉不存在的待删除 Topic
    if (topicsToBeDeleted.size > 0) {
        // 检查待删除 Topic 是否处于不可删除的情况
        topicsToBeDeleted.foreach { topic =>
            // 检测 Topic 中是否有分区正在进行“优先副本”选举
            val preferredReplicaElectionInProgress =
                controllerContext.partitionsUndergoingPreferredReplicaElection
                    .map(_._topic).contains(topic)
            // 检测 Topic 中是否有分区正在进行副本重新分配
            val partitionReassignmentInProgress =
                controllerContext.partitionsBeingReassigned.keySet
                    .map(_._topic).contains(topic)
            if (preferredReplicaElectionInProgress || partitionReassignmentInP
                rogress)
                // 将 Topic 标记为不可删除
                controller.deleteTopicManager.markTopicIneligibleForDeletion(Set
                    (topic))
        }
        // 将可删除的 Topic 提交给 TopicDeletionManager 执行删除操作
        controller.deleteTopicManager.enqueueTopicsForDeletion(topicsToBeDel
            eted)
    }
}

```

PartitionModificationsListener

在上一节介绍的 Topic 删除过程中涉及 PartitionModificationsListener 的注册和取消。在新增 Topic 时会为每个 Topic 注册一个 PartitionModificationsListener，在成功删除 Topic 之后会将注册的 PartitionModificationsListener 删除。PartitionModificationsListener 会监听“/brokers/topics/[topic_name]”节点中的数据变化，主要用于监听一个 Topic 的分区变化。

PartitionModificationsListener.handleDataChange() 方法的实现如下：


```

def handleDataChange(dataPath: String, data: Object) {
    ..... // 加锁和异常处理部分代码(略)
    // 从 ZooKeeper 中获取 Topic 的 Partition 记录
    val partitionReplicaAssignment = zkUtils.getReplicaAssignmentForTopics(List(topic))
    // 过滤出新增分区记录
    val partitionsToBeAdded = partitionReplicaAssignment.filter(p =>
        !controllerContext.partitionReplicaAssignment.contains(p._1))
    // 如果 Topic 正在进行删除, 则输出错误日志后返回
    if (controller.deleteTopicManager.isTopicQueuedUpForDeletion(topic))
        .....// 日志操作(略)
    else {
        if (partitionsToBeAdded.size > 0) {
            // 将新增分区信息添加到 ControllerContext 中
            controllerContext.partitionReplicaAssignment.+=(partitionsToBeAdded)
            // 切换新增分区及其副本的状态, 最终使其上线对外提供服务
            controller.onNewPartitionCreation(partitionsToBeAdded.keySet.toSet)
        }
    }
}

```

onNewPartitionCreation() 方法在 TopicChangeListener 中分析过, 此处不再赘述。需要注意, PartitionModificationsListener 并不对分区的删除进行处理, 在第 5 章中分析 kafka-topics 脚本时可以看到, 是不能减少 Topic 的分区数量的。

BrokerChangeListener

BrokerChangeListener 是 ReplicaStateMachine 中唯一的 ZooKeeper Listener, 它会监听 “/brokers/ids” 节点下的子节点变化, 主要负责处理 Broker 的上线和故障下线。当 Broker 上线时会在 “/brokers/ids” 下创建临时节点, 下线时会删除对应的临时节点。

BrokerChangeListener.handleChildChange() 方法的实现如下:

```

def handleChildChange(parentPath: String, currentBrokerList: java.util.
List[String]) {
    // 省略部分加锁、边界判断和异常处理代码
    // 获取 ZooKeeper 中的 Broker 列表
    val curBrokers = currentBrokerList.map(_.toInt).toSet.flatMap(zkUtils.
getBrokerInfo)
    val curBrokerIds = curBrokers.map(_.id)
    val liveOrShuttingDownBrokerIds = controllerContext.liveOrShuttingDownBrokerIds
    // 过滤, 得到新增的 Broker 列表
    val newBrokerIds = curBrokerIds -- liveOrShuttingDownBrokerIds
    val newBrokers = curBrokers.filter(broker => newBrokerIds(broker.id))
    // 过滤, 得到故障的 Broker 列表
    val deadBrokerIds = liveOrShuttingDownBrokerIds -- curBrokerIds
    // 更新 ControllerContext 的可用 Broker 列表
    controllerContext.liveBrokers = curBrokers
    val newBrokerIdsSorted = newBrokerIds.toSeq.sorted
    val deadBrokerIdsSorted = deadBrokerIds.toSeq.sorted
    val liveBrokerIdsSorted = curBrokerIds.toSeq.sorted
    // 创建 Controller 与新增 Broker 的网络连接
    newBrokers.foreach(controllerContext.controllerChannelManager.addBroker)
    // 关闭 Controller 与故障 Broker 的网络连接
    deadBrokerIds.foreach(controllerContext.controllerChannelManager.
removeBroker)
    // 调用 onBrokerStartup() 方法和 onBrokerFailure() 方法分别处理新增 Broker 和下线
    // Broker
    if (newBrokerIds.size > 0)
        controller.onBrokerStartup(newBrokerIdsSorted)
    if (deadBrokerIds.size > 0)
        controller.onBrokerFailure(deadBrokerIdsSorted)
}

```

KafkaController.onBrokerFailure() 方法对故障 Broker 的处理步骤如下:

(1) 将 Leader 副本分布在故障 Broker 上的分区转换为 OfflinePartition 状态。

(2) 将 OfflinePartition 状态的分区转换为 OnlinePartition 状态。此步会使用 OfflinePartitionLeaderSelector 为其选取 Leader 副本和 ISR 集合并写入 ZooKeeper, 之后发送 LeaderAndIsrRequest 和 UpdateMetadataRequest。

(3) 将故障 Broker 上的副本转换为 OfflineReplica 状态。此步会向故障 Broker 发送 StopReplicaRequest, 从 ISR 集合中清除相关副本, 并发送 LeaderAndIsrRequest 和 UpdateMetadataRequest。

(4) 检查故障 Broker 上是否存在待删除 Topic 的副本, 如果存在, 则将其副本转换为 ReplicaDeletionIneligible 状态并标记 Topic 不可删除。

(5) 如果步骤 1 中没有分区的 Leader 副本在故障 Broker 上, 则上述步骤中可能不会发送 UpdateMetadataRequest, 这里向可用 Broker 发送 UpdateMetadataRequest。

KafkaController.onBrokerFailure() 方法的具体实现如下:

```
def onBrokerFailure(deadBrokers: Seq[Int]) {
  // 将正在正常关闭的 Broker 从 deadBrokers 列表中移除
  val deadBrokersThatWereShuttingDown = deadBrokers
    .filter(id => controllerContext.shuttingDownBrokerIds.remove(id))
  val deadBrokersSet = deadBrokers.toSet
  // 步骤 1: 过滤得到 Leader 副本在故障 Broker 上的分区, 将其转换为 OfflinePartition 状态
  val partitionsWithoutLeader = controllerContext.partitionLeadershipInfo
    .filter(partitionAndLeader =>
      deadBrokersSet.contains(partitionAndLeader._2.leaderAndIsr.leader) &&
        !deleteTopicManager.isTopicQueuedUpForDeletion(partitionAndLeader._1.topic))
    .keySet
  partitionStateMachine.handleStateChanges(partitionsWithoutLeader,
    OfflinePartition)

  // 步骤 2: 将 OfflinePartition 状态的分区转换为 OnlinePartition 状态
  partitionStateMachine.triggerOnlinePartitionStateChange()
  // 步骤 3: 过滤得到在故障 Broker 上的副本, 将这些副本转换为 OfflineReplica 状态
  var allReplicasOnDeadBrokers = controllerContext.replicasOnBrokers(deadBrokersSet)
  val activeReplicasOnDeadBrokers = allReplicasOnDeadBrokers.filterNot(
    p => deleteTopicManager.isTopicQueuedUpForDeletion(p.topic))
  replicaStateMachine.handleStateChanges(activeReplicasOnDeadBrokers,
    OfflineReplica)
```



```
// 步骤4: 检查故障 Broker 上是否有待删除 Topic 的副本, 如果存在, 则将其转换为
// ReplicaDeletionIneligible 状态并标记 Topic 不可删除
val replicasForTopicsToBeDeleted = allReplicasOnDeadBrokers.filter(
  p => deleteTopicManager.isTopicQueuedUpForDeletion(p.topic))
if (replicasForTopicsToBeDeleted.size > 0) {
  deleteTopicManager.failReplicaDeletion(replicasForTopicsToBeDeleted)
}
// 步骤5: 发送 UpdateMetadataRequest 更新所有 Broker 的信息
if (partitionsWithoutLeader.isEmpty) {
  sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.
toSeq)
}
}
```

这里举例说明 `onBrokerFailure()` 方法的功能: 现在假设 Broker0 上分布了 Partition0、1、2 的各个副本, 其中 Partition0 的副本为 Leader 副本, 其余两个副本在对应的 ISR 集合中。当 Broker0 发生故障下线时, ZooKeeper 中的 “/brokers/ids/0” 临时节点会被删除, 并触发 `BrokerChangeListener` 进行处理。首先, 将 Broker0 从 `ControllerContext` 的 Broker 列表中删除。然后, 将 Partition0 转换为 `OfflinePartition` 状态, 紧接着再将其转换成 `OnlinePartition` 状态, 此时会使用 `OfflinePartitionLeaderSelector` 为其选举新的 Leader 副本和 ISR 集合并更新到 ZooKeeper 中, 随后发送 `LeaderAndIsrRequest` 和 `UpdateMetaRequest`。之后, 将三个副本转换成 `OfflineReplica`, 并将其从 ISR 集合删除, 此时会发送 `StopReplicaRequest` (不删除副本)、`LeaderAndIsrRequest` 和 `UpdateMetaRequest` 更新可用 Broker 的 `MetadataCache`。

`KafkaController.onBrokerStartup()` 方法的实现如下:

```
def onBrokerStartup(newBrokers: Seq[Int]) {
  val newBrokersSet = newBrokers.toSet

  // 步骤1: 向集群中所有 Broker 发送 UpdateMetadataRequest, 发送的是所有分区的信息
  // 通过此请求, 集群中所有 Broker 可以了解到新添加的 Broker 信息
  sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.
toSeq)
  // 步骤2: 将新增 Broker 上的副本转换为 OnlineReplica, 此步骤会涉及发送
  // LeaderAndIsrRequest 和 UpdateMetadataRequest
  val allReplicasOnNewBrokers = controllerContext.replicasOnBrokers(newBro
kersSet)
```



```

    replicaStateMachine.handleStateChanges(allReplicasOnNewBrokers,
OnlineReplica)

// 步骤 3: 将 NewPartition 和 OfflinePartition 状态的分区转换成 OnlinePartition 状态
partitionStateMachine.triggerOnlinePartitionStateChange()
// 步骤 4: 检测进行副本重新分配
val partitionsWithReplicasOnNewBrokers = controllerContext
    .partitionsBeingReassigned.filter {
        case (topicAndPartition, reassignmentContext) =>
            reassignmentContext.newReplicas.exists(newBrokersSet.contains(_))
    }
partitionsWithReplicasOnNewBrokers.foreach(p => onPartitionReassignment(p._1,
p._2))
// 步骤 5: 如果新增 Broker 上有待删除 Topic 的副本, 则唤醒 DeleteTopicsThread 线程
// 进行删除
val replicasForTopicsToBeDeleted = allReplicasOnNewBrokers.filter(
    p => deleteTopicManager.isTopicQueuedUpForDeletion(p.topic))
if (replicasForTopicsToBeDeleted.size > 0) {
    deleteTopicManager.resumeDeletionForTopics(
        replicasForTopicsToBeDeleted.map(_.topic))
}
}

```

继续上面的示例, 当 Broker0 重新上线后会创建临时节点 “/brokers/ids/0”, 触发 BrokerChangeListener 处理。首先同样是更新 ControllerContext 的 Broker 列表。然后将三个副本转换成 OnlineReplica 状态, 此时每个分区都已经有了 Leader 副本和 ISR 集合信息, 所以向对应副本发送 LeaderAndIsrRequest 使其成为 Follower, 并向可用 Broker 发送 UpdateMetadataRequest 更新 MetadataCache 信息。此示例没有 Partition 处于 OfflinePartition 状态, 也没有需要进行重新分配的 Partition, 所以后续步骤没有执行。

IsrChangeNotificationListener

在前面介绍过, Follower 副本会与 Leader 副本会进行消息同步, 当 Follower 副本追上 Leader 副本时会被添加到 ISR 集合中, 当 Follower 副本与 Leader 副本差距太大时会被踢出 ISR 集合。Leader 副本不仅会在 ISR 集合变化时将其记录到 ZooKeeper 中, 还会调用 ReplicaManager.recordIsrChange() 方法记录到 isrChangeSet 集合中, 之后通过 isr-change-propagation 定时任务将该集合中周期性地写入到 ZooKeeper 的 “/isr_change_notification”

路径下。KafkaController 中定义的 IsrChangeNotificationListener 用于监听此路径下的子节点变化，当某些分区的 ISR 集合变化时通知整个集群中的所有 Broker。

IsrChangeNotificationListener.handleChildChange() 方法的具体实现如下：

```
override def handleChildChange(parentPath: String,
                                currentChildren: util.List[String]): Unit =
{
    // 省略异常处理部分的代码
    val childrenAsScala: mutable.Buffer[String] = currentChildren.asScala
    val topicAndPartitions: immutable.Set[TopicAndPartition] =
        childrenAsScala.map(x => getTopicAndPartition(x)).flatten.toSet
    if (topicAndPartitions.nonEmpty) {
        // 从 ZooKeeper 中读取指定分区的 Leader 副本、ISR 集合等信息，更新 ControllerContext
        controller.updateLeaderAndIsrCache(topicAndPartitions)
        // 向可用 Broker 发送 UpdateMetadataRequest，更新它们的 MetadataCache
        processUpdateNotifications(topicAndPartitions)
    }
    // 删除 “/isr_change_notification/partitions” 下已经处理的信息
    childrenAsScala.map(x => controller.controllerContext.zkUtils.deletePath(...))
}
```

PreferredReplicaElectionListener

PreferredReplicaElectionListener 负责监听的 ZooKeeper 节点是 “/admin/preferred_replica_election”。当我们通过 PreferredReplicaLeaderElectionCommand 命令指定某些分区需要进行“优先副本”选举时会将指定分区的信息写入该节点，从而触发 PreferredReplicaElectionListener 进行处理。进行“优先副本”选举的目的是让分区的“优先副本”重新成为 Leader 副本，这是为了让 Leader 副本在整个集群中分布得更加均衡。

PreferredReplicaElectionListener.handleDataChange() 方法的具体实现如下：

```
def handleDataChange(dataPath: String, data: Object) {
    inLock(controllerContext.controllerLock) {
        // 获取需要进行“优先副本”选举的 TopicAndPartition 列表
        val partitionsForPreferredReplicaElection = PreferredReplicaLeaderElectionCommand
            .parsePreferredReplicaElectionData(data.toString)
```

```

// 过滤掉正在进行 " 优先副本 " 选举的分区
val partitions = partitionsForPreferredReplicaElection --
    controllerContext.partitionsUndergoingPreferredReplicaElection
// 过滤掉待删除 Topic 的分区
val partitionsForTopicsToBeDeleted = partitions.filter(
    p => controller.deleteTopicManager.isTopicQueuedUpForDeletion(p.
topic))

// 对剩余的分区调用 onPreferredReplicaElection() 方法进行 " 优先副本 " 选举
controller.onPreferredReplicaElection(partitions --
partitionsForTopicsToBeDeleted)
}
}

```

onPreferredReplicaElection() 方法的核心是通过 PreferredReplicaPartitionLeaderSelector 选举 Leader 副本和 ISR 集合。PreferredReplicaPartitionLeaderSelector 选举成功的条件是“优先副本”不是当前 Leader 副本，但是要在 ISR 集合中，否则会抛出异常。经过选举后的 ISR 集合还是当前的 ISR 集合，但分区的 Leader 副本变成了“优先副本”。

```

def onPreferredReplicaElection(partitions: Set[TopicAndPartition],
                                isTriggeredByAutoRebalance: Boolean =
false) {
    // 省略异常处理的相关代码
    // 将参与“优先副本”选举的分区添加到
    // partitionsUndergoingPreferredReplicaElection 集合中
    controllerContext.partitionsUndergoingPreferredReplicaElection +=
partitions
    // 将对应的 Topic 标记为“不可删除”
    deleteTopicManager.markTopicIneligibleForDeletion(partitions.map(_
topic))
    // 将 Partition 转换成 OnlinePartition 状态，除了重选 Leader，还会更新 ZooKeeper
    // 中的数据，并发送 LeaderAndIsrRequest 和 UpdateMetadataRequest
    partitionStateMachine.handleStateChanges(partitions, OnlinePartition,
preferredReplicaPartitionLeaderSelector)
    // 清理 partitionsUndergoingPreferredReplicaElection 集合和 ZooKeeper 上的相
    // 关数据
    removePartitionsFromPreferredReplicaElection(partitions, isTriggeredByAut
oRebalance)
}

```

```
// 将 Topic 标记为“可删除”，并唤醒 DeleteTopicsThread 线程
deleteTopicManager.resumeDeletionForTopics(partitions.map(_.topic))
}
```

onPreferredReplicaElection() 方法还会被一个名为 "partition-rebalance" 的定时任务调用，此任务会定期检测集群中“优先副本”与 Leader 副本的分配情况，并判断是否触发“优先副本”选举，后面会详细介绍该定时任务。

副本重新分配的相关 Listener

PartitionsReassignedListener 监听的 ZooKeeper 节点是 “/admin/reassign_partitions”。当管理人员通过 ReassignPartitionsCommand 命令指定某些分区需要重新分配副本时，会将指定分区的信息写入该节点，从而触发 PartitionsReassignedListener 进行处理。

下面是整个副本重新分配的步骤：

- (1) 从 ZooKeeper 的 “/admin/reassign_partitions” 节点下读取分区重分配信息。
- (2) 过滤掉正在进行重新分配的分区。
- (3) 检测其 Topic 是否为待删除的 Topic。如果是，则调用 KafkaController.removePartitionFromReassignedPartitions() 方法，其操作如下：
 - a) 取消此分区注册的 ReassignedPartitionsIsrChangeListener。
 - b) 删除 ZooKeeper 的 “/admin/reassign_partitions” 节点中与当前分区相关的数据。
 - c) 从 partitionsBeingReassigned 集合中删除分区相关的数据。
- (4) 否则，创建 ReassignedPartitionsContext，调用 initiateReassignReplicasForTopicPartition() 方法开始为重新分配副本做一些准备工作。
 - a) 首先，获取当前的旧 AR 集合和指定的新 AR 集合。
 - b) 比较新旧两个 AR 集合，若两者完全一样则抛出异常，执行步骤 3 的操作后结束。
 - c) 判断新 AR 集合中涉及的 Broker 是否都是可用的，若不是，也抛出异常，执行步骤 3 的操作后结束。
 - d) 为分区添注册 ReassignedPartitionsIsrChangeListener，后面会详细介绍此 Listener。
 - e) 将分区添加到 partitionsBeingReassigned 集合中，并标识该 Topic 不能被删除。
 - f) 调用 onPartitionReassignment() 方法，开始执行副本重新分配。

(5) `onPartitionReassignment()` 方法完成了副本重新分配的整个流程。在下面的描述中使用“新 AR+ 旧 AR”表示新 AR 集合和旧 AR 集合的并集,“新 AR- 旧 AR”表示新 AR 集合与旧 AR 集合的差集。

(6) 判断新 AR 集合中的所有副本是否已经进入了 ISR 集合。如果没有,则执行下面的步骤:

a) 将分区在 `ContextController` 和 `ZooKeeper` 中的 AR 集合更新成“新 AR+ 旧 AR”。

b) 向“新 AR+ 旧 AR”发送 `LeaderAndIsrRequest`, 此步骤主要目的是为了增加 `ZooKeeper` 中记录的 `leader_epoch` 值。

c) 将“新 AR- 旧 AR”中的副本更新成 `NewReplica` 状态, 此步骤会向这些副本发送 `LeaderAndIsrRequest` 使其成为 `Follower` 副本, 并发送 `UpdateMetadataRequest`。

(7) 如果新 AR 集合中的副本已经都进入了 ISR 集合, 则执行下面的步骤:

a) 将新 AR 集合中的所有副本都转换成 `OnlineReplica` 状态。

b) 将 `ControllerContext` 中的 AR 记录更新为新 AR 集合。

c) 如果当前 Leader 副本在新 AR 集合中, 则递增 `ZooKeeper` 和 `ControllerContext` 中记录的 `leader_epoch` 值, 并发送 `LeaderAndIsrRequest` 和 `UpdateMetadataRequest`。

d) 如果当前 Leader 不在新 AR 集合中或 Leader 副本不可用, 则将分区状态转换为 `OnlinePartition` (之前也是 `OnlinePartition`), 主要目的使用 `ReassignedPartitionLeaderSelector` 选举新的 Leader 副本, 使得新 AR 集合中的一个副本成为新 Leader 副本, 然后会发送 `LeaderAndIsrRequest` 和 `UpdateMetadataRequest`。

e) 将“旧 AR- 新 AR”中的副本转换成 `OfflineReplica`, 此步骤会发送 `StopReplicaRequest` (不删除副本), 清理 ISR 集合中的相关副本, 并发送 `LeaderAndIsrRequest` 和 `UpdateMetadataRequest` 请求。

f) 接着将“旧 AR- 新 AR”中的副本转换成 `ReplicaDeletionStarted`, 此步骤会发送 `StopReplicaRequest` (删除副本)。完成删除后, 将副本转换为 `ReplicaDeletionSuccessful`, 最终转换成 `NonExistentReplica`。

g) 更新 `ZooKeeper` 中记录的 AR 信息。

h) 将此分区的相关信息从 `ZooKeeper` 的“`/admin/reassign_partitions`”节点中移除。

i) 向所有可用的 Broker 发送一次 `UpdateMetadataRequest`。

j) 尝试取消相关的 Topic 的“不可删除”标记，并唤醒 DeleteTopicsThread 线程。

上面描述的整个流程其实还涉及 ReassignedPartitionsIsrChangeListener 的相关内容，其中，步骤 7 就是在 ReassignedPartitionsIsrChangeListener 中完成的。

ReassignedPartitionsIsrChangeListener 在上述步骤 (4) → d 中注册到 ZooKeeper 的 “/broker/topics/[topic_name]/partitions/[partitionId]/state” 节点上监听其数据变化，主要负责处理进行副本重新分配的分区 ISR 集合变化。当 ReassignedPartitionsIsrChangeListener 监听到分区的 ISR 集合发生变化时，按照下列步骤进行处理：

(1) 检查当前分区是否正在进行副本的重新分配操作，若不是，则结束。

(2) 从 ZooKeeper 中读取当前分区的 Leader 和 ISR 集合。

(3) 如果新 AR 集合中的副本已完全进入当前 ISR 集合，则调用 onPartitionReassignment() 方法完成步骤 7 的相关操作。

(4) 否则，输出日志后结束，等待下一次触发。

为了让读者更好地理解副本重新分配的过程，通过一个示例描述这个过程：现在假设有 Broker0~5 这六个 Broker，Topic 名为“test”的编号为 0 的分区 (Partition0) 三个副本，分别位于 Broker1~3 上，其中 Leader 副本位于 Broker1 上。

管理人员使用 ReassignPartitionsCommand 命令将 Partition0 的副本重新分配到 Broker3~5 上。PartitionsReassignedListener 被触发后首先为 Partition0 注册 ReassignedPartitionsIsrChangeListener，并标记 test 这个 Topic 不能被删除。然后，将 ZooKeeper 和 ControllerContext 中 Partition0 的 AR 集合记录更新为 [1,2,3,4,5]。发送 LeaderAndIsrRequest 使 [4,5] 副本成为 Follower 副本，并发送 UpdateMetadataRequest 更新可用 Broker 的 MetadataCache。到这里 PartitionsReassignedListener 的相关处理就结束了。

随着 [4,5] 两个 Follower 副本与 Leader 副本进行同步，最终会进入 ISR 集合，此时会触发 ReassignedPartitionsIsrChangeListener 进行处理。首先将 [3,4,5] 副本转换为 OnlineReplica 状态，更新 ControllerContext 中对应的 AR 记录为 [3,4,5]。之后，[3,4,5] 副本已经处于 ISR 集合，且 Leader 副本不在 [3,4,5] 中，则需要使用 ReassignedPartitionLeaderSelector 选举新的 Leader。这里我们简单回顾 ReassignedPartitionLeaderSelector 的策略，它选取的新 Leader 副本必须在新指定的 AR 集合中且在 ISR 集合中，当前 ISR 集合为新 ISR 集合。接收 LeaderAndIsrRequest 的副本是新指定的 AR 集合。这里假设选举副本 3 为 Leader 副本，并向 [3,4,5] 三个副本发送 LeaderAndIsrRequest 请求。然后，将 [1,2] 副本转换为 OfflineReplica，此时发送 StopReplicaRequest 停止副本，将其从 ISR 中移除，最终将

这两个副本删除转换为 NonExistentReplica 状态。最后，更新 ZooKeeper 中的 AR 记录，标记取消“test”的“不可删除”标记，并唤醒 DeleteTopicsThread 线程。副本重新分配到此全部完成，此时 AR 集合和 ISR 集合都为 [3,4,5]，Leader 为副本 3。

PartitionsReassignedListener.handleDataChange() 方法的具体实现如下：

```
def handleDataChange(dataPath: String, data: Object) {
    // 省略一些加锁和异常处理的操作
    // 步骤 1: 从 ZooKeeper 的 “/admin/reassign_partitions” 节点下读取分区的副本重分
    // 配信息
    val partitionsReassignmentData = zkUtils
        .parsePartitionReassignmentData(data.toString)
    // 步骤 2: 过滤掉正在进行重新分配的分区
    val partitionsToBeReassigned = partitionsReassignmentData.filterNot(p =>
        controllerContext.partitionsBeingReassigned.contains(p._1))
    partitionsToBeReassigned.foreach { partitionToBeReassigned =>
        // 步骤 3: 检测其 Topic 是否为待删除 Topic
        if (controller.deleteTopicManager.isTopicQueuedUpForDeletion(
            partitionToBeReassigned._1.topic)){
            controller.removePartitionFromReassignedPartitions(partitionToBeReas
signed._1)
        } else {
            val context = new ReassignedPartitionsContext(partitionToBeReassign
ed._2)
            // 步骤 4: 为副本重新分配做一些准备工作
            controller.initiateReassignReplicasForTopicPartition(...)
        }
    }
}
```

KafkaController.initiateReassignReplicasForTopicPartition() 方法的实现如下：

```
def initiateReassignReplicasForTopicPartition(topicAndPartition: TopicAndPartition,
        reassignedPartitionContext:
ReassignedPartitionsContext) {
    ... .. // 省略异常处理的部分代码，步骤 4.a
    if (assignedReplicas == newReplicas) {
        ... .. // 步骤 4.b: 新旧副本分配方式完全一样，则抛出异常
    } else {
```

```

if (aliveNewReplicas == newReplicas) { // 步骤 4.c: 判断新 AR 是否都可用
    // 步骤 4.d: 为分区注册 ReassignedPartitionsIsrChangeListener
    watchIsrChangesForReassignedPartition(topic,
        partition, reassignedPartitionContext)
    controllerContext.partitionsBeingReassigned.put(topicAndPartition,
        reassignedPartitionContext)
    // 步骤 4.e: 将 Topic 标记为“不可删除”
    deleteTopicManager.markTopicIneligibleForDeletion(Set(topic))
    // 步骤 4.f: 执行副本的重新分配
    onPartitionReassignment(topicAndPartition, reassignedPartitionContext)
}
}
}

```

`KafkaController.onPartitionReassignment()` 是副本重新分配的核心，其具体实现如下：

```

def onPartitionReassignment(topicAndPartition: TopicAndPartition,
                            reassignedPartitionContext:
ReassignedPartitionsContext) {
    val reassignedReplicas = reassignedPartitionContext.newReplicas
    areReplicasInIsr(topicAndPartition.topic, topicAndPartition.partition,
reassignedReplicas) match {
        case false =>
            val newReplicasNotInOldReplicaList = reassignedReplicas.toSet --
                controllerContext.partitionReplicaAssignment(topicAndPartition).
toSet
            val newAndOldReplicas = (reassignedPartitionContext.newReplicas ++
                controllerContext.partitionReplicaAssignment(topicAndPartition)).
toSet
            // 步骤 6.a: 将 Partition 在 ContextController 和 ZooKeeper 中的 AR 集合更新成
            // “新 AR+ 旧 AR”
            updateAssignedReplicasForPartition(topicAndPartition, newAndOldReplic
as.toSeq)
            // 步骤 6.b: 通过发送 LeaderAndIsrRequest, 递增 leader_epoch
            updateLeaderEpochAndSendRequest(topicAndPartition,
                controllerContext.partitionReplicaAssignment(topicAndPartition),
                newAndOldReplicas.toSeq)
            // 步骤 6.c: 将“新 AR- 旧 AR”中的副本更新成 NewReplica 状态

```



```

        startNewReplicasForReassignedPartition(topicAndPartition,
            reassignedPartitionContext, newReplicasNotInOldReplicaList)
    case true =>
        // 获取旧 AR 集合
        val oldReplicas = controllerContext
            .partitionReplicaAssignment(topicAndPartition).toSet --
            reassignedReplicas.toSet
        // 步骤 7.a: 将新 AR 集合中的所有副本都转换成 OnlineReplica 状态
        reassignedReplicas.foreach { replica =>
            replicaStateMachine.handleStateChanges(Set(new PartitionAndReplica(
                topicAndPartition.topic, topicAndPartition.partition, replica)),
                OnlineReplica)
        }
        // 步骤 7.b~d
        moveReassignedPartitionLeaderIfRequired(topicAndPartition,
            reassignedPartitionContext)
        // 步骤 7.e~f: 将“旧 AR- 新 AR”中的副本转换成 OfflineReplica
        stopOldReplicasOfReassignedPartition(topicAndPartition,
            reassignedPartitionContext, oldReplicas)
        // 步骤 7.g: 更新 ZooKeeper 中记录的 AR 信息
        updateAssignedReplicasForPartition(topicAndPartition, reassignedReplicas)
        // 步骤 7.h: 将此 Partition 的相关信息从 ZooKeeper 的
        // “/admin/reassign_partitions” 节点中移除
        removePartitionFromReassignedPartitions(topicAndPartition)
        controllerContext.partitionsBeingReassigned.remove(topicAndPartition)
        // 步骤 7.i: 向所有可用的 Broker 发送一次 UpdateMetadataRequest。
        sendUpdateMetadataRequest(controllerContext
            .liveOrShuttingDownBrokerIds.toSeq, Set(topicAndPartition))
        // 步骤 7.j: 尝试取消相关的 Topic 的“不可删除”标记, 并唤醒 DeleteTopicsThread
        deleteTopicManager.resumeDeletionForTopics(Set(topicAndPartition.
            topic))
    }
}

```

ReassignedPartitionsIsrChangeListener 的核心逻辑也是调用 onPartitionReassignment() 方法, 这里不再赘述了。

4.6.8 KafkaController 初始化与故障转移

KafkaController 的主要功能和实现在前面几节中已经介绍完了。本节介绍 KafkaController 的初始化以及故障转移方面的内容。在 Kafka 集群中，只有一个 Controller 能够成为 Leader 来管理整个集群，而其他未成为 Controller Leader 的 Broker 上也会创建一个 KafkaController 对象，它们唯一能做的事情就是当 Controller Leader 出现故障，不能继续管理集群时，竞争成为新的 Controller Leader。

KafkaController 的启动和故障转移的过程与 ZookeeperLeaderElector 有着密切的关系，KafkaController.controllerElector 字段 ZookeeperLeaderElector 类型的定义如下：

```
private val controllerElector = new ZookeeperLeaderElector(controllerContext,
    zkUtils.ControllerPath, onControllerFailover,
    onControllerResignation, config.brokerId)
```

KafkaController 启动的过程由 KafkaController.startup() 方法完成，其中会注册 SessionExpirationListener，并启动 ZookeeperLeaderElector。

```
def startup() = {
    inLock(controllerContext.controllerLock) {
        registerSessionExpirationListener() // 注册 SessionExpirationListener
        isRunning = true
        controllerElector.startup // 启动 ZookeeperLeaderElector
    }
}
```

SessionExpirationListener 继承了 IZkStateListener 接口，监听 KafkaController 与 ZooKeeper 的连接状态。当 KafkaController 与 ZooKeeper 的连接超时后创建新连接时会触发 SessionExpirationListener.handleNewSession() 方法。

```
def handleNewSession() {
    inLock(controllerContext.controllerLock) {
        onControllerResignation() // 负责清理 KafkaController 依赖的对象
        controllerElector.elect // 尝试选举新 Controller Leader
    }
}
```

KafkaController 的启动逻辑委托给了 ZooKeeperLeaderElector.startup() 方法完成。在

ZookeeperLeaderElector 中有两个比较重要的字段：

- leaderId: 缓存当前的 Controller LeaderId。
- leaderChangeListener: LeaderChangeListener 会 监 听 Zookeeper 的 “/controller” 节点的数据变化，当此节点中保存的 LeaderId 发生变化时，会触发 LeaderChangeListener 进行相应的处理。

```
def handleDataChange(dataPath: String, data: Object) {
  inLock(controllerContext.controllerLock) {
    val amILeaderBeforeDataChange = amILeader
    // 记录新 Controller Leader 的 id
    leaderId = KafkaController.parseControllerId(data.toString)
    // 如果当前 Broker 由 Controller Leader 变成 Follower，则要进行相应的清理工作
    if (amILeaderBeforeDataChange && !amILeader)
      // 旧 Controller Leader 的清理工作，通过 ZookeeperLeaderElector 的构造参数
      // 我们知道，这里调用的 onResigningAsLeader()
      // 实际是 KafkaController.onControllerResignation()
      onResigningAsLeader()
  }
}
```

当 “/controller” 节点中的数据被删除时会触发 handleDataDeleted() 方法进行处理。

```
def handleDataDeleted(dataPath: String) {
  ... .. // Debug 信息输出
  if (amILeader)
    // 这里调用的 onResigningAsLeader() 实际是 KafkaController.onControllerResignation()
    onResigningAsLeader()
  elect // 尝试新 Controller Leader 的选举，下面详述
}
```

ZookeeperLeaderElector.startup() 方法的逻辑是注册 LeaderChangeListener 后，立即调用 elect() 方法尝试进行 Controller Leader 的选举。


```
def startup {
  inLock(controllerContext.controllerLock) {
    // 在 ZooKeeper 的 “/controller” 节点上注册 LeaderChangeListener 进行监听
    controllerContext.zkUtils.zkClient.subscribeDataChanges(electionPath,
      leaderChangeListener)
    elect // 进行选举
  }
}
```

KafkaController 触发选举的地方有三处，如图 4-65 所示。依次是：第一次启动的时候；LeaderChangeListener 监听到 “/controller” 节点中数据被删除；ZooKeeper 连接过期并重新建立之后。

- ➔ 1 ZookeeperLeaderElector.elect() (kafka.server)
- ▶ ① ZookeeperLeaderElector.startup() (kafka.server)
- ▶ ① LeaderChangeListener in ZookeeperLeaderElector.handleDataDeleted(String) (kafka.server)
- ▶ ① SessionExpirationListener in KafkaController.handleNewSession() (kafka.controller)

图 4-65

ZookeeperLeaderElector.elect() 方法的具体实现如下：

```
def elect: Boolean = {
  leaderId = getControllerID // 获取当前 ZooKeeper 中记录的 Controller Leader
  的 id
  if (leaderId != -1) { // 已存在 Controller Leader，放弃选举
    return amILeader
  }
  try {
    // 尝试创建临时节点，如果临时节点已经存在，则抛出异常
    val zkCheckedEphemeral = new ZKCheckedEphemeral(electionPath,
      electString, controllerContext.zkUtils.zkConnection.getZookeeper,
      JaasUtils.isZkSecurityEnabled())
    zkCheckedEphemeral.create()

    leaderId = brokerId // 更新 leaderId 字段，当前 Broker 成功成为 Controller
    Leader
  }
```



```

// 通过 ZookeeperLeaderElector 的构造参数我们知道，这里调用的
// onBecomingLeader() 实际是 KafkaController.onControllerFailover() 方法
onBecomingLeader()
} catch {
    ... ..// 异常处理（略）
// 对 onBecomingLeader() 方法抛出异常的处理，重置 leaderId，并删除 “/controller” 路径
resign()
}
amILeader // 检测当前的 Broker 是否为 Controller Leader
}

```

onControllerFailover

ZookeeperLeaderElector elect() 方法中调用的回调函数 onBecomingLeader() 实际上是 ZookeeperLeaderElector 构造函数中传入的 KafkaController.onControllerFailover() 方法。当前 Broker 成功选举为 Controller Leader 时会通过该方法完成一系列的初始化操作，其具体步骤如下：

```

def onControllerFailover() {
    // 省略部分边界检查和日志输出代码（略）
    // 步骤 1：读取 Zookeeper 中记录的 ControllerEpochPath 信息并更新到
    // ControllerContext 中
    readControllerEpochFromZookeeper() // 具体路径是 “/controller_epoch”
    // 步骤 2：递增 ControllerEpochPath，并写入 ZooKeeper 中
    incrementControllerEpoch(zkUtils.zkClient)
    // 步骤 3：注册上文介绍的一系列 ZooKeeper 监听器
    registerReassignedPartitionsListener() // 注册 PartitionsReassignedListener
    registerIsrChangeNotificationListener() // 注册 IsrChangeNotificationListener
    registerPreferredReplicaElectionListener() // 注册 PreferredReplicaElecti
onListener
    // 注册 TopicChangeListener、DeleteTopicsListener
    partitionStateMachine.registerListeners()
    replicaStateMachine.registerListeners() // 注册 BrokerChangeListener
    // 步骤 4：初始化 ControllerContext，主要是从 ZooKeeper 中读取 Topic、分区、副本相关
    // 的各种元数据信息，并且启动 ControllerChannelManager、TopicDeletionManager
    // 等依赖组件
}

```

```

initializeControllerContext()
// 步骤 5: 启动 ReplicaStateMachine 组件, 其中会初始化各个副本的状态
replicaStateMachine.startup()
// 步骤 6: 启动 PartitionStateMachine 组件, 其中会初始化各个分区的状态
partitionStateMachine.startup()
// 步骤 7: 为所有 Topic 注册 PartitionModificationsListener
controllerContext.allTopics.foreach(topic =>
    partitionStateMachine.registerPartitionChangeListener(topic))
// 修改 Broker 的状态
brokerState.newState(RunningAsController)
// 步骤 8: 处理副本重新分配的分区, 调用 initiateReassignReplicasForTopicPartition()
// 方法实现, 前面已介绍, 不再赘述
maybeTriggerPartitionReassignment()
// 步骤 9: 处理需要进行“优先副本”选举的分区, 调用 onPreferredReplicaElection()
// 方法实现, 前面已介绍, 不再赘述
maybeTriggerPreferredReplicaElection()
// 步骤 10: 向集群中所有的 Broker 发送 UpdateMetadataRequest 更新其 MetadataCache
sendUpdateMetadataRequest(controllerContext.liveOrShuttingDownBrokerIds.
    toSeq)
// 步骤 11: 根据配置决定开启分区的自动均衡功能
if (config.autoLeaderRebalanceEnable) {
    autoRebalanceScheduler.startup() // 启动定时任务, 周期性检测
    autoRebalanceScheduler.schedule("...", checkAndTriggerPartitionRebalance, ...)
}
// 步骤 12: 启动 TopicDeletionManager, 底层启动 DeleteTopicsThread
deleteTopicManager.start()
}

```

此过程的组件我们在前面已经详细分析过了, 这里我们关注一下 `initializeControllerContext()` 方法, 了解 `ControllerContext` 到底从 `ZooKeeper` 中读取了哪些数据。

```

private def initializeControllerContext() {
    // update controller cache with delete topic information
    // 下面是从 zk 中初始化一堆东西...初始化 controllerContext 的字段...

    // 读取 “/brokers/ids”，初始化可用 Broker 集合
    controllerContext.liveBrokers = zkUtils.getAllBrokersInCluster().toSet
    // 读取 “/brokers/topics”，初始化集群中全部的 Topic 信息
    controllerContext.allTopics = zkUtils.getAllTopics().toSet
    // 读取 “/brokers/topics/[topic_name]/partitions”，初始化每个 Partition 的
    // AR 集合信息
    controllerContext.partitionReplicaAssignment = zkUtils
        .getReplicaAssignmentForTopics(controllerContext.allTopics.toSeq)
    controllerContext.partitionLeadershipInfo =
        new mutable.HashMap[TopicAndPartition, LeaderIsrAndControllerEpoch]
    controllerContext.shuttingDownBrokerIds = mutable.Set.empty[Int]
    // 读取 “/brokers/topics/[topic_name]/partitions/[partitionId]/state”，
    // 初始化每个 Partition 的 Leader、ISR 集合等信息
    updateLeaderAndIsrCache()
    // 启动 ControllerChannelManager
    startChannelManager()
    // 读取 “/admin/preferred_replica_election”，初始化需要“优先副本”选举的
    // Partition
    initializePreferredReplicaElection()
    // 读取 “/admin/reassign_partitions”，初始化需要进行副本重新分配的 Partition
    initializePartitionReassignment()
    // 启动 TopicDeletionManager
    initializeTopicDeletion()
}

```

Partition Rebalance

在 `KafkaController.onControllerFailover()` 方法中会启动一个名为“partition-rebalance”的周期性的定时任务，它提供了分区的自动均衡功能。该定时任务会周期性地调用 `KafkaController.checkAndTriggerPartitionRebalance()` 方法对失衡的 Broker 上相关的分区进行“优先副本”选举，使得相关分区的“优先副本”重新成为 Leader 副本，整个集群中 Leader 副本的分布也会重新恢复平衡。

`checkAndTriggerPartitionRebalance()` 方法首先获取“优先副本”所在的 `BrokerId` 与分

区的对应关系，然后利用此对应关系，计算每个“优先副本”所在的 Broker 的“imbalance”比率，该值是当前 Leader 副本为非“优先副本”的分区与此 Broker 上分区总数的比值，当“imbalance”比率大于一定阈值，则触发“优先副本”选举。

为了读者更好地理解含义，现举例说明。假设集群中有 0 ~ 2 三个 Broker，某 Topic 有 15 个分区，Leader 副本即为“优先副本”，它们的分布如表 4-6 所示。

表 4-6

Broker0	Broker1	Broker2
Partition1	Partition2	Partition3
Partition4	Partition5	Partition6
Partition7	Partition8	Partition9
Partition10	Partition11	Partition12
Partition13	Partition14	Partition15

当 Kafka 集群运行一段时间，期间某些 Broker 可能出现过宕机，导致 Leader 副本发生迁移，现在 Leader 副本的分布如表 4-7 所示。

表 4-7

Broker0	Broker1	Broker2
Partition1	Partition2	Partition3
Partition4		Partition6
Partition7		Partition9
Partition10		Partition12
Partition13		Partition15
Partition5		Partition11
Partition8		Partition14

此时的 Leader 副本分布已经明显不均匀了，Broker2 上分布了 5 个分区的副本，但当前有 4 个分区并不是以“优先副本”为 Leader 副本，Broker2 的“imbalance”比率为 $4/5=80\%$ ，默认的阈值为 10%，此时会触发对 Partition5、Partition8、Partition11、Partition14 的“优先副本”选举。

KafkaController.checkAndTriggerPartitionRebalance() 方法的实现如下：


```

private def checkAndTriggerPartitionRebalance(): Unit = {
    // 获取所有可用的 Broker 的副本
    var preferredReplicasForTopicsByBrokers:
        Map[Int, Map[TopicAndPartition, Seq[Int]]] = null
    // 获取“优先副本”所在的 BrokerId 与分区的对应关系
    preferredReplicasForTopicsByBrokers =
        controllerContext.partitionReplicaAssignment.filterNot(
            p => deleteTopicManager.isTopicQueuedUpForDeletion(p._1.topic))
            .groupBy {
                case (topicAndPartition, assignedReplicas) => assignedReplicas.
head
            }

    // 计算每个 Broker 的“imbalance”比率
    preferredReplicasForTopicsByBrokers.foreach {
        case (leaderBroker, topicAndPartitionsForBroker) => {
            var imbalanceRatio: Double = 0
            var topicsNotInPreferredReplica: Map[TopicAndPartition, Seq[Int]] = null
            // 存在 Leader 副本，但不是以“优先副本”为 Leader 的分区集合
            topicsNotInPreferredReplica =
                topicAndPartitionsForBroker.filter {
                    case (topicPartition, replicas) => {
                        controllerContext.partitionLeadershipInfo.
contains(topicPartition) &&
                        controllerContext.partitionLeadershipInfo(topicPartition)
                            .leaderAndIsr.leader != leaderBroker
                    }
                }
            val totalTopicPartitionsForBroker = topicAndPartitionsForBroker.size
            val totalTopicPartitionsNotLedByBroker = topicsNotInPreferredReplica.
size
            // 计算当前 Broker 的“imbalance”比率
            imbalanceRatio = totalTopicPartitionsNotLedByBroker.toDouble /
                totalTopicPartitionsForBroker

            // Broker 上的“imbalance”比率大于一定阈值时，触发“优先副本”选举

```



```

def onControllerResignation() {
    // 取消 ZooKeeper 上的监听器
    deregisterIsrChangeNotificationListener() // 取消 IsrChangeNotificationListener
    deregisterReassignedPartitionsListener() // 取消 ReassignedPartitionsListener
    deregisterPreferredReplicaElectionListener() // 取消 PreferredReplicaElectionListener

    if (deleteTopicManager != null) // 关闭 TopicDeletionManager
        deleteTopicManager.shutdown()

    // 停止 partition-rebalance 定时任务
    if (config.autoLeaderRebalanceEnable)
        autoRebalanceScheduler.shutdown()

    // 取消所有的 ReassignedPartitionsIsrChangeListener
    deregisterReassignedPartitionsIsrChangeListeners()
    // 关闭 PartitionStateMachine 和 ReplicaStateMachine
    partitionStateMachine.shutdown()
    replicaStateMachine.shutdown()

    // 关闭 ControllerChannelManager, 断开与集群中其他 Broker 的连接
    controllerContext.controllerChannelManager.shutdown()
    ... ..
    brokerState.newState(RunningAsBroker) // 切换 Broker 状态
}

```

4.6.9 处理 ControlledShutdownRequest

在前面介绍的 BrokerChangeListener 可以根据 ZooKeeper 中 “/brokers/ids” 的子节点变化, 处理 Broker 故障宕机的场景。在有些场景中用户希望主动关闭正常运行的 Broker, 例如, 更换硬件、操作系统升级、修改 Kafka 的配置等。如果依然使用上述方式关闭 Broker, 就略显粗暴。在 Kafka 中提供了 Controlled Shutdown 的方式来关闭一个 Broker 实例。

使用 Controlled Shutdown 的方式主动下线 Broker 有两个好处: 一是可以让日志文件完全同步到磁盘上, 在 Broker 下次重现上线时不需要进行 Log 的恢复操作; 二是 Controller Shutdown 方式在关闭 Broker 之前, 会对其上的 Leader 副本进行迁移, 这样就

可以减少分区的不可用时间。

在 Kafka 之前的版本中需要使用命令行工具向 Controller Leader 发送 ControlledShutdownRequest 请求。在本书分析的版本中, 已经将该过程写成了 JVM 的关闭钩子。在 Kafka.main() 方法中可以看到这段代码:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
    override def run() = {
        // 在 kafkaServerStartable.shutdown() 方法中会首先调用 controlledShutdown()
        // 方法向 Controller Leader 发送 ControlledShutdownRequest, 具体逻辑后面详细介绍
        kafkaServerStartable.shutdown
    }
})
```

KafkaController.shutdownBroker() 方法是 ControlledShutdownRequest 的核心, 该方法会使用 ControlledShutdownLeaderSelector 重新选择 Leader 副本和 ISR 集合, 实现 Leader 副本的迁移。回顾一下 ControlledShutdownLeaderSelector 的策略: 从当前 ISR 集合中排除正在关闭的副本后的剩余副本作为新 ISR 集合, 从新 ISR 集合中选择新的 Leader, 需要向可用的 AR 发送 LeaderAndIsrRequest。shutdownBroker() 方法的代码如下所示。

```
def shutdownBroker2(id: Int): Set[TopicAndPartition] = {
    // 省略部分边界检查、加锁以及异常处理代码
    // 步骤 1: 获取待关闭 Broker 上所有的 Partition 和副本信息
    val allPartitionsAndReplicationFactorOnBroker: Set[(TopicAndPartition,
    Int)] =
        controllerContext.partitionsOnBroker(id)
            .map(topicAndPartition => (topicAndPartition,
                controllerContext.partitionReplicaAssignment(topicAndPartition).
                size))

    allPartitionsAndReplicationFactorOnBroker.foreach {
        case (topicAndPartition, replicationFactor) =>
            // Move leadership serially(连续地) to relinquish(放弃, 交出) lock...
            controllerContext.partitionLeadershipInfo.get(topicAndPartition)
                .foreach { currLeaderIsrAndControllerEpoch =>
                    if (replicationFactor > 1) { // 步骤 2: 是否开启副本机制
                        // 步骤 3: 检测 Leader 副本是否处于待关闭 Broker 上
                        if (currLeaderIsrAndControllerEpoch.leaderAndIsr.leader == id) {
                            // 步骤 4: 将相关的 Partition 切换为 OnlinePartition 状态。此步骤使用
```



```

        // ControlledShutdownLeaderSelector 重新选择 Leader 和 ISR 集合, 并将
        // 结果写入 ZooKeeper, 之后发送 LeaderAndIsrRequest 和 UpdateMetadataRequest
        partitionStateMachine.handleStateChanges(Set(topicAndPartition),
            OnlinePartition, controlledShutdownPartitionLeaderSelector)
    } else {
        try {
            // 步骤 4: 发送 StopReplicaRequest (不删除副本)
            brokerRequestBatch.newBatch()
            brokerRequestBatch.addStopReplicaRequestForBrokers(Seq(id),
                topicAndPartition.topic, topicAndPartition.partition,
                deletePartition = false)
            brokerRequestBatch.sendRequestsToBrokers(epoch)
        } catch {
            // 异常处理 (略)
        }
        // 步骤 4: 将副本转换成 OfflineReplica 状态
        replicaStateMachine.handleStateChanges(
            Set(PartitionAndReplica(topicAndPartition.topic,
                topicAndPartition.partition, id)), OfflineReplica)
    }
}

}

}

}

// 统计 Leader 副本依然处于待关闭 Broker 上的分区, 并将其返回
def replicatedPartitionsBrokerLeads() = inLock(controllerContext.
controllerLock) {
    trace("All leaders = " + controllerContext.partitionLeadershipInfo.
mkString(", "))
    controllerContext.partitionLeadershipInfo.filter {
        case (topicAndPartition, leaderIsrAndControllerEpoch) =>
            leaderIsrAndControllerEpoch.leaderAndIsr.leader == id &&
            controllerContext.partitionReplicaAssignment(topicAndPartition).
size > 1
    }.map(_._1)
}
replicatedPartitionsBrokerLeads().toSet
}

```

在本节中我们介绍了 ContextController 如何管理集群相关的各种信息，包括 Broker、Topic、分区、副本等元数据，以及读取、更新 ZooKeeper 的时机。了解了 ControllerChannelManager 和 ControllerBrokerRequestBatch 如何帮助 KafkaController 完成与集群中其他的 Broker 的通信，以及 LeaderAndIsrRequest 和 UpdateMetadataRequest 等请求的格式。介绍了管理分区状态的 PartitionStateMachine 组件和管理副本状态的 ReplicaStateMachine，以及在不同场景下使用的 PartitionLeaderSelector 策略。通过注册不同多种不同类型的 ZooKeeper Listener，可以实现多种管理功能，例如：Topic 增删的相关处理、Broker 故障下线的处理、分区的副本重新分配、“优先副本”选举，等等。我们还介绍了一个 Broker 如何成为 Controller Leader，成为 Controller Leader 后的初始化流程以及放弃 Controller Leader 角色后的清理操作。最后，又分析了 Controlled Shutdown 方式下一个 Broker 的具体实现。希望读者通过本节的阅读，能够对 KafkaController 的功能和实现有清晰的了解。

4.7 GroupCoordinator

第3章中详细分析了新版消费者的工作原理和具体实现，与新版消费者密切相关的服务端组件是 GroupCoordinator。如图 4-66 所示，在每一个 Broker 上都会实例化一个 GroupCoordinator 对象，Kafka 按照 Consumer Group 的名称将其分配给对应的 GroupCoordinator 进行管理；每个 GroupCoordinator 只负责管理 Consumer Group 的一个子集，而非集群中全部的 Consumer Group。请读者注意与 KafkaController 以及副本机制中的主从模式的区别。

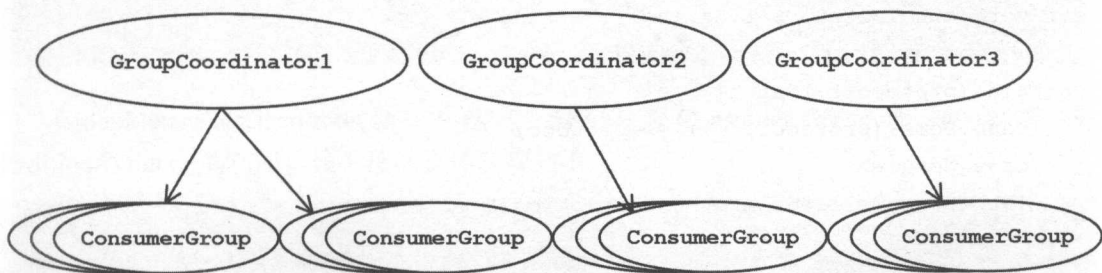


图 4-66

GroupCoordinator 有几项比较重要的功能：一是负责处理 JoinGroupRequest 和 SyncGroupRequest 完成 Consumer Group 中分区的分配工作；二是通过 GroupMetadataManager 和内部 Topic “Offsets Topic” 维护 offset 信息，即使出现消费者宕机也可以找回之前提交的 offset；三是记录 Consumer Group 的相关信息，即使 Broker 宕

机导致 Consumer Group 由新的 GroupCoordinator 进行管理, 新 GroupCoordinator 也可以知道 Consumer Group 中每个消费者负责处理哪个分区等信息; 四是通过心跳消息检测消费者的状态。

GroupCoordinator 中使用 MemberMetadata 记录消费者的元数据, MemberMetadata 中各字段的含义和功能如下所述。

- memberId: 对应消费者的 id, 此值是由服务端的 GroupCoordinator 分配的。
- groupId: 记录消费者所在的 Consumer Group 的 id。
- assignment: Array[Byte] 类型, 记录了分配给当前 Member 的分区信息。
- supportedProtocols: 对应消费者支持的 PartitionAssignor。在第 3 章已经介绍了与 PartitionAssignor 的相关内容, 请读者回顾。
- awaitingJoinCallback: 与 JoinGroupRequest 相关的回调函数, 后面详述。
- awaitingSyncCallback: 与 SyncGroupRequest 相关的回调函数, 后面详述。
- sessionTimeoutMs: 心跳超时时间。
- latestHeartbeat: 最后一次收到心跳消息的时间戳。
- isLeaving: 标识对应消费者是否已经离开了 Consumer Group。

MemberMetadata.vote() 方法提供了从给定候选 PartitionAssignor 中选择消费者支持的 PartitionAssignor 的功能, 该方法实现如下:

```
def vote(candidates: Set[String]): String = {
    supportedProtocols.find({case (protocol, _) => candidates.
contains(protocol)})match{
        case Some((protocol, _)) => protocol
        case None =>
            throw new IllegalArgumentException("...")
    }
}
```

GroupMetadata 记录了 Consumer Group 的元数据信息, 其字段的含义如下所述。

- groupId: 对应 Consumer Group 的 id。
- members: HashMap[String, MemberMetadata] 类型, key 是 memberId, value 是对应的 MemberMetadata 对象。

- state: GroupState 类型, 标识当前 Consumer Group 所处的状态。
- generationId: 标识当前 Consumer Group 的年代信息, 避免受到过期请求的影响。
- leaderId: 记录 Consumer Group 中的 Leader 消费者的 memberId。
- protocol: 记录了当前 Consumer Group 选择的 PartitionAssignor。

在 GroupMetadata 中提供了对上述字段的操作, 例如对 members 集合的增删、对 state 的切换。这里有个细节需要读者注意, GroupMetadata 在进行 Member 的增删操作时, 还会顺便选择 Group Leader:

```
def add(memberId: String, member: MemberMetadata) {
  if (leaderId == null)
    leaderId = memberId // 第一个加入的 Member 即为 Group Leader
  members.put(memberId, member)
}

def remove(memberId: String) {
  members.remove(memberId)
  if (memberId == leaderId) {
    leaderId = if (members.isEmpty) {
      null
    } else {
      members.keys.head // Group Leader 被删除, 则重新选择 Group Leader
    }
  }
}
```

GroupMetadata.selectProtocol() 方法实现了为 Consumer Group 选择合适的 PartitionAssignor 的功能, 该方法的具体实现如下:

```
def selectProtocol: String = {
  // 所有 Member 都支持的协议作为“候选协议”集合
  val candidates = allMemberMetadata.map(_.protocols)
    .reduceLeft((commonProtocols, protocols) => commonProtocols &
    protocols)
  // 每个 Member 都会通过 vote() 方法进行投票, 每个 Member 会为其 supportedProtocols
  // 集合中第一个“候选协议”投一票, 最终将选择得票最多的 PartitionAssignor
```



```
val votes: List[(String, Int)] = allMemberMetadata.map(_.vote(candidates))
    .groupBy(identity).mapValues(_.size).toList
votes.maxBy(_._2)._1
}
```

GroupCoordinator 除了管理 Consumer Group 分配信息, 还记录了每个 Consumer Group 的 offset 信息。GroupCoordinator 使用 GroupTopicPartition 维护 Consumer Group 与分区的消费关系, 使用 OffsetAndMetadata 记录 offset 的相关信息。

```
case class OffsetAndMetadata(offsetMetadata: OffsetMetadata,
    commitTimestamp: Long, expireTimestamp: Long) {... ...}

// metadata 默认为空字符串
case class OffsetMetadata(offset: Long, metadata: String = OffsetMetadata.NoMetadata)

case class GroupTopicPartition(group: String, topicPartition: TopicPartition)
{... ...}
```

4.7.1 GroupMetadataManager

GroupMetadataManager 是 GroupCoordinator 中负责管理 Consumer Group 元数据以及其对应 offset 信息的组件。GroupMetadataManager 底层使用 Offsets Topic, 以消息的形式存储 Consumer Group 的 GroupMetadata 信息以及其消费的每个分区的 offset, 如图 4-67 所示。



图 4-67

为了提高查询的效率, GroupMetadataManager 同时还会将 Consumer Group 的

GroupMetadata 信息和 offset 信息在内存中维护一份相同的副本, 并进行同步修改。

GroupMetadataManager 依赖的组件如图 4-68 所示。

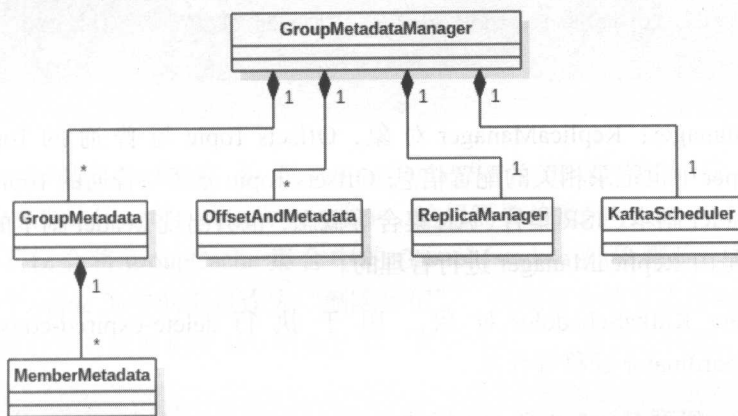


图 4-68

GroupMetadataManager 中各个字段的含义和功能如下所述。

- offsetsCache: Pool[GroupTopicPartition, OffsetAndMetadata] 类型, 记录了每个 Consumer Group 消费的分区的 offset 位置。
- groupsCache: Pool[String, GroupMetadata] 类型, 记录每个 Consumer Group 在服务端对应的 GroupMetadata 对象。
- loadingPartitions: Set[Int] 类型, 记录了正在加载的 Offsets Topic 分区的 id, 后续详述。
- ownedPartitions: Set[Int] 类型, 记录了已经加载的 Offsets Topic 分区的 id, 后续详述。
- groupMetadataTopicPartitionCount: 记录 Offsets Topic 的分区数量。该字段会调用 getOffsetsTopicPartitionCount() 方法进行初始化, 如下:

```
// 使用 getOffsetsTopicPartitionCount() 方法进行初始化
private val groupMetadataTopicPartitionCount = getOffsetsTopicPartitionCount

private def getOffsetsTopicPartitionCount = {
  val topic = TopicConstants.GROUP_METADATA_TOPIC_NAME
  // 从 ZooKeeper 中获取 Offsets Topic 这个内部 Topic 的分区信息和副本信息
  val topicData = zkUtils.getPartitionAssignmentForTopics(Seq(topic))
}
```

```

if (topicData(topic).nonEmpty)
  topicData(topic).size // 返回分区数量
else
  config.offsetsTopicNumPartitions // 返回的配置数量
}

```

- **replicaManager**: **ReplicaManager** 对象, **Offsets Topic** 与普通的 **Topic** 一样, 在 **ZooKeeper** 中也记录相关的配置信息; **Offsets Topic** 分区与普通的 **Topic** 的分区一样, 也有 **Leader** 副本、**ISR** 集合、**AR** 集合等概念, 也会出现 **Leader** 副本的迁移等情况, 所以也是由 **ReplicaManager** 进行管理的。
- **scheduler**: **KafkaScheduler** 对象, 用于执行 **delete-expired-consumer-offsets**、**GroupCoordinator** 迁移等任务。

groupsCache 管理与 offsetsCache 管理

记录 **GroupMetadata** 信息的信息和记录消费 **offset** 位置的消息都是通过 **partitionFor()** 方法在 **Offsets Topic** 中选择合适的分区。

```

// 对 groupId 的 hash 值取模, 得到 Consumer Group 对应的 Offsets Topic 分区编号
def partitionFor(groupId: String): Int =
  Utils.abs(groupId.hashCode) % groupMetadataTopicPartitionCount

```

所以同一 **Consumer Group** 对应的这两类消息会被分配到同一个 **Offsets Topic** 分区中, 但是这两类消息的 **key** 有所不同:

```

// 下面是 GroupMetadataManager.groupMetadataKey() 方法的实现, 该方法用于创建记录
// GroupMetadata 的消息的 key, 仅由 groupId 这一个字段组成
private def groupMetadataKey(group: String): Array[Byte] = {
  val key = new Struct(CURRENT_GROUP_KEY_SCHEMA)
  key.set(GROUP_KEY_GROUP_FIELD, group)
  ..... // 将 key 写入到 ByteBuffer 并返回 (略)
}

// 下面是 GroupMetadataManager.offsetCommitKey() 方法的实现, 该方法用于创建记录
// GroupMetadata 的消息的 key, 由 groupId、Topic 名称、PartitionId 三部分组成
private def offsetCommitKey(group: String, topic: String, partition: Int,
  versionId: Short = 0): Array[Byte] = {

```

```

val key = new Struct(CURRENT_OFFSET_KEY_SCHEMA)
key.set(OFFSET_KEY_GROUP_FIELD, group)
key.set(OFFSET_KEY_TOPIC_FIELD, topic)
key.set(OFFSET_KEY_PARTITION_FIELD, partition)
..... // 将 key 写入到 ByteBuffer 并返回 (略)
}

```

GroupMetadataManager 提供了对 groupsCache 集合的管理方法, getGroup()、addGroup() 方法实现比较简单, 请读者参考源码, 这里需要注意的是 removeGroup() 方法。removeGroup() 不仅会将 groupCache 集合中的 GroupMetadata 对象删除, 还会向 Offsets Topic 中写入一个 value 为空的消息作为“删除标记”。在前面介绍日志压缩时提到过, 将 value 为空的消息看作对前面相同 key 的消息的“删除标记”, 请读者回顾。

```

def removeGroup(group: GroupMetadata) {
    // 删除 groupCache 中的 GroupMetadata
    if (groupsCache.remove(group.groupId, group)) {
        // 获取 Consumer Group 在 Offsets Topic 中对应的分区的 id
        val groupPartition = partitionFor(group.groupId)
        val (magicValue, timestamp) = getMessageFormatVersionAndTimestamp(groupPartition)
        // 产生“删除标记”消息, 注意, value 是 null, key 由 groupId 封装得来
        val tombstone = new Message(bytes = null,
            key = GroupMetadataManager.groupMetadataKey(group.groupId),
            timestamp = timestamp, magicValue = magicValue)
        // 获取 Offsets Topic 中对应的 Partition 对象
        val partitionOpt = replicaManager.getPartition(TopicConstants.GROUP_METADATA_TOPIC_NAME, groupPartition)
        partitionOpt.foreach { partition =>
            val appendPartition = TopicAndPartition(TopicConstants.GROUP_METADATA_TOPIC_NAME,
                groupPartition)
            // 写入消息, 此处省略异常处理代码
            partition.appendMessagesToLeader(new ByteBufferMessageSet(
                config.offsetsTopicCompressionCodec, tombstone))
        }
    }
}

```


GroupMetadataManager 提供了对 offsetsCache 集合的管理方法，putOffset()、getOffset() 方法比较简单，请读者参考源码，但是没有提供直接对 offsetsCache 集合进行删除的方法，而是使用定时任务的方式周期性地清理 offsetCache 集合。在 GroupMetadataManager 初始化时会创建并启动一个 KafkaScheduler 对象，然后启动一个名为“delete-expired-consumer-offsets”的定时任务，它会周期性地调用 deleteExpiredOffsets() 方法。

```
private val scheduler = new KafkaScheduler(threads = 1,
    threadNamePrefix = "group-metadata-manager-")
scheduler.startup() // 启动 KafkaScheduler

// 启动“delete-expired-consumer-offsets”这个定时任务
scheduler.schedule(name = "delete-expired-consumer-offsets",
    fun = deleteExpiredOffsets(),
    period = config.offsetsRetentionCheckIntervalMs,
    unit = TimeUnit.MILLISECONDS)
```

GroupMetadataManager.deleteExpiredOffsets() 方法的删除原理与 removeGroup() 有些类似，除了需要删除 offsetsCache 集合中对应的 OffsetMetadata 对象，还需要向 Offsets Topic 中追加“删除标记”消息，但步骤略复杂一点。具体代码分析如下：

```
private def deleteExpiredOffsets() {
    val numExpiredOffsetsRemoved = inWriteLock(offsetExpireLock) {
        // 过滤得到所有过期的 OffsetAndMetadata
        val expiredOffsets = offsetsCache.filter {
            case (groupTopicPartition, offsetAndMetadata) =>
                offsetAndMetadata.expireTimestamp < startMs
        }
        val tombstonesForPartition = expiredOffsets.map {
            case (groupTopicAndPartition, offsetAndMetadata) =>
                // 找到该 groupid 对应的存储 offset 消息的 Offsets Topic 分区的 id
                val offsetsPartition = partitionFor(groupTopicAndPartition.group)
                // 删除对应的 OffsetAndMetadata 对象
                offsetsCache.remove(groupTopicAndPartition)
                // 获取消息的 key
                val commitKey = GroupMetadataManager.offsetCommitKey(
                    groupTopicAndPartition.group,
                    groupTopicAndPartition.topicPartition.topic,
```

```

        groupTopicAndPartition.topicPartition.partition)
    // 获取对应分区使用的魔数和时间戳
    val (magicValue, timestamp) = getMessageFormatVersionAndTimestamp(
        offsetsPartition)
    // 返回的是对应 Partition 的 Id 和一个“删除标记”消息
    (offsetsPartition, new Message(bytes = null,
        key = commitKey, timestamp = timestamp, magicValue = magicValue))
}.groupBy { case (partition, tombstone) => partition }
// Offsets Topic 的一个分区中可能记录了多个 Consumer Group 的 offset 信息
// 为了便于追加“删除标记”消息，上面最后按照 Offsets Topic 的分区 id 进行了一次分组

tombstonesForPartition.flatMap { case (offsetsPartition, tombstones) =>
    val partitionOpt = replicaManager.getPartition(
        TopicConstants.GROUP_METADATA_TOPIC_NAME, offsetsPartition)
    partitionOpt.map { partition => // 获取对应的 Offsets Topic 分区
        // 获取要追加到此 Offsets Topic 分区中的“删除标记”消息集合
        val messages = tombstones.map(_._2).toSeq
        // 追加“删除标记”消息
        partition.appendMessagesToLeader(new ByteBufferMessageSet(
            config.offsetsTopicCompressionCodec, messages: _*))
        tombstones.size
    }
}.sum
}
}
}

```

查找 GroupCoordinator

消费者与 GroupCoordinator 交互之前，首先会发送 GroupCoordinatorRequest 到负载较小的 Broker，目的是查询管理其所在 Consumer Group 对应的 GroupCoordinator 的网络位置。之后，消费者会连接到 GroupCoordinator，发送剩余的 JoinGroupRequest 和 SyncGroupRequest。

KafkaApis.handleGroupCoordinatorRequest() 负责 GroupCoordinatorRequest 的相关处理。首先使用 partitionFor() 方法得到负责保存对应 Consumer Group 信息的 Offsets Topic 分区，然后查找其 MetadataCache，得到此 Offsets Topic 分区的 Leader 副本所在的 Broker，其上的 GroupCoordinator 负责管理该 Consumer Group。虽然不是 GroupCoordinator 中的方法，但是了解此过程，有助于理解 Consumer Group 与 GroupCoordinator 之间对应关系的确定。

方式。

```
def handleGroupCoordinatorRequest(request: RequestChannel.Request) {
  // 省略验证部分代码
  // 通过 groupId 得到对应的 Offsets Topic 分区的 id
  val partition = coordinator.partitionFor(groupCoordinatorRequest.
groupId)
  // 从 MetadataCache 中获取 Offsets Topic 的相关信息, 如果 Offsets Topic 还未创建,
  // 则会在这里创建, 具体创建过程后面介绍
  val offsetsTopicMetadata = getOrCreateGroupMetadataTopic(request.
securityProtocol)
  // 通过上述 Offsets Topic Partition 的 Id 获取其 leader 所在的 Node
  val coordinatorEndpoint = offsetsTopicMetadata.partitionMetadata().
asScala
    .find(_.partition == partition).map(_.leader())
  coordinatorEndpoint match {
    // 创建 GroupCoordinatorResponse
    case Some(endpoint) if !endpoint.isEmpty =>
      new GroupCoordinatorResponse(Errors.NONE.code, endpoint)
    case _ =>
      new GroupCoordinatorResponse(Errors.GROUP_COORDINATOR_NOT_AVAILABLE.
code,
        Node.noNode)
  }
  // 将响应加入 RequestChannel, 等待发送
  requestChannel.sendResponse(new RequestChannel.Response(request,
    new ResponseSend(request.connectionId, responseHeader, responseBody)))
}
```

通过上面的描述, 我们确定了负责管理 Consumer Group 的 GroupCoordinator。我们可以认为 Consumer Group 与 Offsets Topic 分区之间的关系是多对一的, 因为多个 Consumer Group 的 GroupMetadata 信息和 Offset 信息会以消息的方式存放于一个 Offsets Topic 分区中; Offsets Topic 分区与 GroupCoordinator 之间的关系也是多对一的, 因为多个 Offsets Topic 分区的 Leader 可以位于一个 Broker 之上, 如图 4-69 所示。

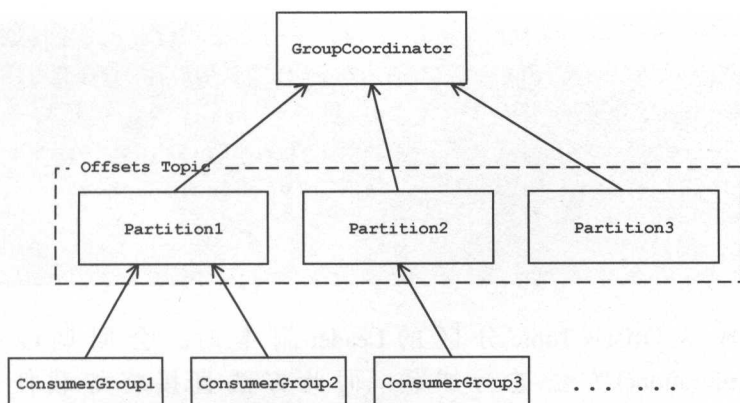


图 4-69

GroupCoordinator 迁移

默认配置下，Offsets Topic 有 50 个分区，每个分区有 3 个副本。根据上一节介绍，当某个 Leader 副本所在的 Broker 出现故障时会发生迁移，那么 Consumer Group 则由新 Leader 副本所在的 Broker 上运行的 GroupCoordinator 负责管理，GroupCoordinator 应该如何将 groupsCache 和 offsetsCache 集合中的信息转移到新 GroupCoordinator 上呢？

这部分功能是在 `KafkaApis.handleLeaderAndIsrRequest()` 方法中实现的。在该方法处理完 `LeaderAndIsrRequest` 之后，会回调 `onLeadershipChange()` 方法完成 GroupCoordinator 迁移操作，其实现如下：

```

def handleLeaderAndIsrRequest(request: RequestChannel.Request) {
  // 这里关注 onLeadershipChange() 方法，省略了其他部分的代码
  // 下面是 onLeadershipChange() 方法的定义
  def onLeadershipChange(updatedLeaders: Iterable[Partition],
                          updatedFollowers: Iterable[Partition]) {
    updatedLeaders.foreach { partition =>
      if (partition.topic == TopicConstants.GROUP_METADATA_TOPIC_NAME)
        coordinator.handleGroupImmigration(partition.partitionId)
    }
    updatedFollowers.foreach { partition =>
      if (partition.topic == TopicConstants.GROUP_METADATA_TOPIC_NAME)
        coordinator.handleGroupEmigration(partition.partitionId)
    }
  }
}
... ..

```



```

// onLeadershipChange() 方法作为 ReplicaManager.becomeLeaderOrFollower() 方法
// 的回调函数传入，在 becomeLeaderOrFollower() 方法的最后一行会调用
// onLeadershipChange() 方法
val result = replicaManager.becomeLeaderOrFollower(correlationId,
    leaderAndIsrRequest, metadataCache, onLeadershipChange)
... ..
}

```

当 Broker 成为 Offsets Topic 分区的 Leader 副本时，会回调 GroupCoordinator.handleGroupImmigration() 方法进行加载，而此方法直接将加载任务委托给了 GroupCoordinator.loadGroupsForPartition() 方法。在 loadGroupsForPartition() 方法中，通过 KafkaScheduler 以任务的形式调用 loadGroupsAndOffsets() 方法，而当前线程直接返回。具体的加载步骤如下：

(1) 检测当前的 Offsets Topic 分区是否正在加载。如果是，则结束本次加载操作，否则将其加入 loadingPartitions 集合，标识该分区正在进行加载。

(2) 通过 ReplicaManager 组件得到此分区对应的 Log 对象。

(3) 从 Log 对象中的第一个 LogSegment 开始加载，加载过程中可能会碰到记录了 Offset 信息的信息，也有可能碰到记录 GroupMetadata 信息的信息，还有可能是“删除标记”消息，需要区分处理：

a) 如果是记录 Offset 信息的信息且是“删除标记”，则删除 offsetsCache 集合中对应的 OffsetAndMetadata 对象。

b) 如果是记录 Offset 信息的信息且不是“删除标记”，则解析消息形成 OffsetAndMetadata 对象，添加到 offsetsCache 集合中。

c) 如果是记录 GroupMetadata 信息的信息，则统计是否为“删除标记”，在步骤 4 中进行处理。

(4) 根据步骤 (3) → c 中的统计，将需要加载的 GroupMetadata 信息加载到 groupsCache 集合中，并检测需要删除的 GroupMetadata 信息是否还在 groupsCache 集合中。

(5) 将当前 Offset Topic 分区的 id 从 loadingPartitions 集合移入 ownedPartitions 集合，标识该分区加载完成，当前 GroupCoordinator 开始正式负责管理其对应的 Consumer Group。

GroupCoordinator.loadGroupsAndOffsets() 方法的具体代码和注释如下，代码略长但不复杂：

```

def loadGroupsAndOffsets() {
  // 步骤 1: 检测当前 Partition 是否正在加载, 省略加锁代码
  if (loadingPartitions.contains(offsetsPartition)) {
    return
  } else {
    loadingPartitions.add(offsetsPartition)
  }

  try {
    // 步骤 2: 从 LogManager 中得到这个 partition 对应的 Log 实例
    replicaManager.logManager.getLog(topicPartition) match {
      case Some(log) =>
        // 步骤 3: 从 Log 中的第一个 LogSegment 开始加载
        var currOffset = log.logSegments.head.baseOffset
        val buffer = ByteBuffer.allocate(config.loadBufferSize) // 创建缓冲区
        // 加锁, 防止与 deleteExpiredOffsets() 方法发生并发
        inWriteLock(offsetExpireLock) {
          val loadedGroups = mutable.Map[String, GroupMetadata]()
          val removedGroups = mutable.Set[String]()
          // 读取 Log 的结束位置是 HW
          while(currOffset < getHighWatermark(offsetsPartition)&&!shutting
Down.get()) {
            buffer.clear()
            // 读取 Log, 注意, 返回的是分片的 FileMessageSet 对象, 读者请参考 Log 相关的小节
            val messages = log.read(currOffset,
              config.loadBufferSize).messageSet.asInstanceOf[FileMessageSet]
            // 将消息读取到内存, 形成 ByteBufferMessageSet
            messages.readInto(buffer, 0)
            val messageSet = new ByteBufferMessageSet(buffer)
            messageSet.foreach { msgAndOffset => // 迭代消息集合
              val baseKey = GroupMetadataManager.readMessageKey(
                msgAndOffset.message.key)
              if (baseKey.isInstanceOf[OffsetKey]) {
                // 步骤 3.a: 读取到记录 Offset 信息的消息
                val key = baseKey.key.asInstanceOf[GroupTopicPartition]
                // 若是“删除标记”消息, 则删除 offsetsCache 中对应的 OffsetAndMetadata
                if (msgAndOffset.message.payload == null) {
                  if (offsetsCache.remove(key) != null)

```

```

        trace("Removed offset for %s due to tombstone
entry.".format(key))
    else
        trace("Ignoring redundant tombstone for
%s.".format(key))
    } else {
        // 不是“删除标记”消息，则解析 value
        val value = GroupMetadataManager.readOffsetMessageValue(
            msgAndOffset.message.payload)
        // 向 offsetsCache 集合中添加对应的 OffsetAndMetadata，注意 expireTimestamp
        putOffset(key, value.copy(
            expireTimestamp = {
                if (value.expireTimestamp == OffsetCommitRequest.
DEFAULT_TIMESTAMP)
                    value.commitTimestamp + config.offsetsRetentionMs
                else
                    value.expireTimestamp
            }
        ))
        trace("Loaded offset %s for %s.".format(value, key))
    }
} else { // 步骤 3.b: 读取到记录 GroupMetadata 信息的消息
    // 解析 key 和 value
    val groupId = baseKey.key.asInstanceOf[String]
    val groupMetadata = GroupMetadataManager.
readGroupMessageValue(
        groupId, msgAndOffset.message.payload)
    // 根据是否为“删除标记”消息，进行统计
    if (groupMetadata != null) {
        removedGroups.remove(groupId)
        loadedGroups.put(groupId, groupMetadata)
    } else {
        loadedGroups.remove(groupId)
        removedGroups.add(groupId)
    }
}
currOffset = msgAndOffset.nextOffset
}
}

```



```

// 步骤4: 将需要加载的 GroupMetadata 信息加载到 groupsCache 集合中
loadedGroups.values().foreach { group =>
    val currentGroup = addGroup(group)
    if (group != currentGroup)
        // Debug 日志输出 (略)
    else
        onGroupLoaded(group) // 后续描述
}
// 检测需要删除的 GroupMetadata 信息是否还在 groupsCache 集合中
removedGroups.foreach { groupId =>
    val group = groupsCache.get(groupId)
    if (group != null)
        throw new IllegalStateException("...")
}
}
case None => warn("No log found for " + topicPartition)
}
}
catch {
    // 异常处理 (略)
    case t: Throwable =>
        error("Error in loading offsets from " + topicPartition, t)
}
finally {
    // 步骤5: 将当前 Offset Topic 分区从 loadingPartitions 移入 ownedPartitions 集合
    loadingPartitions.synchronized {
        ownedPartitions.add(offsetsPartition)
        loadingPartitions.remove(offsetsPartition)
    }
}
}
}
}

```

当 Broker 成为 Offsets Topic 分区的 Follower 副本时会回调 GroupCoordinator.handleGroupEmigration() 方法进行清除工作。在 handleGroupEmigration() 方法中会调用 removeGroupsForPartition() 方法, 它也是通过 KafkaScheduler 以任务的形式调用 removeGroupsAndOffsets() 方法, 而当前线程直接返回。具体的清理操作步骤如下:

(1) 从 ownedPartitions 集合中将对应的 Offsets Topic 分区删除, 标识当前

GroupCoordinator 不再管理其对应 Consumer Group。

(2) 遍历 offsetsCache 集合，将此分区对应的 OffsetAndMetadata 全部清除。

(3) 遍历 groupsCache 集合，将此分区对应的 GroupMetadata 全部清除。

需要注意的是，这里步骤 2 和步骤 3 仅仅是清空集合，并没有向 Log 中追加“删除标记”消息，因为其他 Broker 会成为此 Offsets Topic 分区的 Leader 副本，还需要使用其中记录的 GroupMetadata 信息和 Offset 信息。removeGroupsAndOffsets() 方法的具体代码如下：

```
def removeGroupsAndOffsets() {
  // 省略统计和日志输出的部分代码
  var numOffsetsRemoved = 0
  var numGroupsRemoved = 0

  loadingPartitions synchronized {
    // 步骤 1: 从 ownedPartitions 集合中将对应的 Offsets Topic 分区删除
    // 标识当前 GroupCoordinator 不再管理其对应 Consumer Group
    ownedPartitions.remove(offsetsPartition)
    // 步骤 2: 遍历 offsetsCache 集合，将该分区对应的 OffsetAndMetadata 全部清除
    offsetsCache.keys.foreach { key =>
      if (partitionFor(key.group) == offsetsPartition) {
        offsetsCache.remove(key)
        numOffsetsRemoved += 1
      }
    }
    // 步骤 3: 遍历 groupsCache 集合，将该分区对应的 GroupMetadata 全部清除
    for (group <- groupsCache.values) {
      if (partitionFor(group.groupId) == offsetsPartition) {
        onGroupUnloaded(group)
        groupsCache.remove(group.groupId, group)
        numGroupsRemoved += 1
      }
    }
  }
}
```

在 GroupMetadataManager 中还提供了两个比较重要的检测方法。第一个是 isGroupLocal() 方法，用于检测当前 GroupCoordinator 是否管理指定的 Consumer Group。

第二个是 `isGroupLoading()` 方法，用于检测指定的 Consumer Group 对应的 Offsets Topic 分区是否还处于上述加载过程之中。

```
def isGroupLocal(groupId: String): Boolean =
  loadingPartitions synchronized ownedPartitions.contains(partitionFor(groupId))

def isGroupLoading(groupId: String): Boolean =
  loadingPartitions synchronized loadingPartitions.contains(partitionFor(groupId))
```

在处理 `JoinGroupRequest`、`OffsetFetchRequest`、`OffsetCommitRequest` 及 `HeartbeatRequest` 这些请求之前，都会调用这两个方法进行检测，如果检测失败，则直接返回异常响应。

SyncGroupRequest 相关处理

在第3章中提到，Consumer Group 中的 Leader 消费者通过 `SyncGroupRequest` 将分区的分配结果发送给 `GroupCoordinator`，`GroupCoordinator` 会根据此分配结果形成 `SyncGroupResponse` 返回给所有消费者。消费者收到 `SyncGroupResponse` 后进行解析，即可得知分区的分配结果。

`GroupCoordinator` 除了会根据分区分配结果发送给所有消费者，还会将其形成消息，追加到对应的 Offsets Topic 分区中。`GroupMetadataManager.prepareStoreGroup()` 方法实现了根据分区分配结果创建消息的功能，其调用栈如图 4-70 所示。

```
▼ ① GroupMetadataManager.prepareStoreGroup(GroupMetadata, Map<String, byte[]>, Function1<Object, BoxedUnit>)
  ▼ ② GroupCoordinator.doSyncGroup(GroupMetadata, int, String, Map<String, byte[]>, Function2<byte[], Object, BoxedUnit>)
    ▼ ③ GroupCoordinator.handleSyncGroup(String, int, String, Map<String, byte[]>, Function2<byte[], Object, BoxedUnit>)
      ► ④ KafkaApis.handleSyncGroupRequest(Request)
```

图 4-70

`GroupMetadataManager.prepareStoreGroup()` 方法的具体实现如下：

```
def prepareStoreGroup(group: GroupMetadata, groupAssignment: Map[String,
Array[Byte]] ,
                      responseCallback: Short => Unit): DelayedStore = {
  // 获取对应 Offsets Topic Partition 使用消息版本格式
  val (magicValue, timestamp) =
    getMessageFormatVersionAndTimestamp(partitionFor(group.groupId))
  // 创建记录 GroupMetadata 信息的消息，消息的 value 是分区的分配结果
  val message = new Message(
```

```

    key = GroupMetadataManager.groupMetadataKey(group.groupId),
    bytes = GroupMetadataManager.groupMetadataValue(group, groupAssignment),
    timestamp = timestamp, magicValue = magicValue)
// 获取 Consumer Group 对应的 Offsets Topic 分区
val groupMetadataPartition = new TopicPartition(
    TopicConstants.GROUP_METADATA_TOPIC_NAME, partitionFor(group.groupId))
// Offsets Topic 分区与消息集合的对应关系
val groupMetadataMessageSet = Map(groupMetadataPartition ->
    new ByteBufferMessageSet(config.offsetsTopicCompressionCodec, message))

// 下面定义的 putCacheCallback() 回调函数会在上述消息成功追加到 Offsets Topic 分区
// 之后被调用
def putCacheCallback(responseStatus: Map[TopicPartition, PartitionResponse]) {
    ... .. // 具体实现后面会分析
}
DelayedStore(groupMetadataMessageSet, putCacheCallback)
}

```

`prepareStoreGroup()` 并没有追加消息的代码，它仅仅是创建了 `DelayedStore` 对象，其中封装了消息和回调函数。真正实现追加消息操作的是 `GroupMetadataManager.store()` 方法，其中会调用 `ReplicaManager.appendMessages()` 方法追加消息，我们这里要注意的是 `appendMessages()` 方法中的参数。第二个参数是 `requiredAcks`，这里的默认值是 -1，即 ISR 集合中所有副本都已经同步了该消息才认为消息成功追加并返回；第三个参数是 `internalTopicsAllowed`，这里的值是 `true`，表示可以向 Kafka 内部 Topic 追加消息，读者可简单回顾 `ReplicaManager.appendToLocalLog()` 方法的实现。

```

def store(delayedAppend: DelayedStore) {
    // 调用 ReplicaManager.appendMessages() 方法追加消息
    replicaManager.appendMessages(
        config.offsetCommitTimeoutMs.toLong, config.offsetCommitRequiredAcks,
        true, delayedAppend.messageSet, delayedAppend.callback)
}

```

当 `requiredAcks` 参数为 -1 时，需要创建 `DelayedProduce` 并等待相应的条件满足后才执行完成并调用回调函数。这里的回调函数就是 `GroupMetadataManager.prepareStoreGroup()` 方法中定义的 `putCacheCallback()` 方法，它的参数是追加消息的结果。


```
def putCacheCallback(responseStatus: Map[TopicPartition, PartitionResponse])
{
    ..... // 边界检查 (略)
    // 下面根据消息追加结果更新错误码
    val status = responseStatus(groupMetadataPartition)
    var responseCode = Errors.NONE.code
    if (status.errorCode != Errors.NONE.code) {
        ..... // 追加消息异常的情况 (略)
    }
    // 此回调函数是 prepareStoreGroup() 方法的第三个参数
    responseCallback(responseCode)
}
```

我们回到 `prepareStoreGroup()` 方法调用的地方, 分析下面的代码片段, 重点关注传入的回调函数:

```
delayedGroupStore = Some(groupManager.prepareStoreGroup(group, assignment,
    (errorCode: Short) => {
        group synchronized {
            // 首先, 检测 Consumer Group 状态以及年代信息
            if (group.is(AwaitingSync) && generationId == group.generationId) {
                if (errorCode != Errors.NONE.code) {
                    .....// 异常场景处理 (略)
                } else {
                    setAndPropagateAssignment(group, assignment) // 重点分析此方法的内容
                    group.transitionTo(Stable) // Consumer Group 的状态在后面描述
                }
            }
        }
    })
}))
```

其中的 `setAndPropagateAssignment()` 方法主要完成三项工作: 一是将分配结果更新到 `GroupMetadata` 维护的每个 `MemberMetadata` 中; 二是调用每个 `MemberMetadata` 的 `awaitingSyncCallback` 回调函数, 该回调函数中的具体操作是创建 `SyncGroupResponse` 对象并添加到 `RequestChannel` 中等待发送; 三是将本次心跳延迟任务完成并开始下次等待心跳的延迟任务的执行或超时。


```

private def setAndPropagateAssignment(group: GroupMetadata,
                                      assignment: Map[String, Array[Byte]])
{
    assert(group.is(AwaitingSync))
    // 第一工作: 更新 GroupMetadata 中每个相关的 MemberMetadata.assignment 字段
    group.allMemberMetadata.foreach(member =>
        member.assignment = assignment(member.memberId))
    // 完成第二项和第三项工作
    propagateAssignment(group, Errors.NONE.code)
}

// 下面是 propagateAssignment() 方法的代码
private def propagateAssignment(group: GroupMetadata, errorCode: Short) {
    for (member <- group.allMemberMetadata) {
        if (member.awaitingSyncCallback != null) {
            // 第二工作: 调用 awaitingSyncCallback 回调函数
            member.awaitingSyncCallback(member.assignment, errorCode)
            member.awaitingSyncCallback = null
            // 第三工作: 开启等待下次心跳的延迟任务
            completeAndScheduleNextHeartbeatExpiration(group, member)
        }
    }
}

```

GroupMetadataManager 中与 SyncGroupRequest 相关的处理到这里就介绍完了。

OffsetCommitRequest 相关处理

消费者在进行正常的消费过程以及 Rebalance 操作之前, 都会进行提交 offset 的操作, 其核心任务是将消费者消费的每个分区对应的 offset 封装成 OffsetCommitRequest 发送给 GroupCoordinator。GroupCoordinator 会将这些 offset 封装成消息, 追加到对应的 Offsets Topic 分区中。

在 GroupMetadataManager 中与 OffsetCommitRequest 处理相关的方法有两个: 一是 GroupMetadataManager.prepareStoreOffsets() 方法, 它负责产生 DelayedStore 对象, 其中封装了待追加的消息集合和追加后需要执行的回调函数; 二是 store() 方法, 它负责向 Offsets Topic 分区中追加消息, 在上一节已经介绍了。prepareStoreOffsets() 方法的调用栈如图 4-71 所示。

```

▼ ① GroupMetadataManager.prepareStoreOffsets(String, String, int, Map<TopicPartition, OffsetAndMetadata>, Function1<
  ▼ ① GroupCoordinator.handleCommitOffsets(String, String, int, Map<TopicPartition, OffsetAndMetadata>, Function1<
    ► ① KafkaApis.handleOffsetCommitRequest(Request) (kafka.server)

```

图 4-71

prepareStoreOffsets() 方法与 prepareStoreGroup() 方法有些类似，实现分析如下：

```

def prepareStoreOffsets(groupId: String, consumerId: String, generationId:
Int,
                                offsetMetadata: immutable.Map[TopicPartition,
OffsetAndMetadata],
                                responseCallback: immutable.Map[TopicPartition,
Short] => Unit): DelayedStore = {
  // 检测 OffsetAndMetadata.metadata 字段的长度，metadata 字段默认是空字段
  // 消费者可以在 OffsetCommitRequest 中携带除 offset 之外的额外说明信息，经过解析后
  // 会添加到 metadata 字段
  val filteredOffsetMetadata = offsetMetadata.filter {
    case (topicPartition, offsetAndMetadata) =>
      validateOffsetMetadataLength(offsetAndMetadata.metadata)
  }
  val messages = filteredOffsetMetadata.map {
    case (topicAndPartition, offsetAndMetadata) =>
      // 获取对应 Offsets Topic 分区使用消息格式信息
      val (magicValue, timestamp) =
        getMessageFormatVersionAndTimestamp(partitionFor(groupId))
      // 创建记录 Offset 信息的消息，消息的 value 是 offsetAndMetadata 中的数据
      new Message(key = GroupMetadataManager.offsetCommitKey(groupId,
        topicAndPartition.topic, topicAndPartition.partition),
        bytes = GroupMetadataManager.offsetCommitValue(offsetAndMetadata),
        timestamp = timestamp,
        magicValue = magicValue
      ).toSeq
  }
  // 获取 Consumer Group 对应的 Offsets Topic 分区
  val offsetTopicPartition = new TopicPartition(
    TopicConstants.GROUP_METADATA_TOPIC_NAME,
    partitionFor(groupId))
  // 获取 Offsets Topic 分区与消息集合的对应关系
  val offsetsAndMetadataMessageSet = Map(offsetTopicPartition ->

```

```

        new ByteBufferMessageSet(config.offsetsTopicCompressionCodec, messages:
_*)))

// 下面的 putCacheCallback() 回调方法根据追加结果设置错误码并调用回调函数,
// 与上述 prepareStoreGroup() 方法中定义的同名方法功能类似
def putCacheCallback(responseStatus: Map[TopicPartition, PartitionResponse]) {
    ... ..// 边界检查 (略)
    // 下面根据消息追加结果更新错误码
    val status = responseStatus(offsetTopicPartition)
    val responseCode = if (status.errorCode == Errors.NONE.code) {
        filteredOffsetMetadata.foreach { case (topicAndPartition,
offsetAndMetadata) =>
            // 追加消息成功, 则更新 offsetsCache 集合中对应的 OffsetAndMetadata 对象
            putOffset(GroupTopicPartition(groupId, topicAndPartition),
offsetAndMetadata)
        }
        Errors.NONE.code
    } else {
        ... ..// 追加消息异常的情况 (略)
    }
    responseCallback(commitStatus) // 调用回调函数
}
// 返回 DelayedStore 对象
DelayedStore(offsetsAndMetadataMessageSet, putCacheCallback)
}

```

prepareStoreOffsets() 方法返回的 DelayedStore 对象经过 store() 方法处理后, 完成消息的追加并调用回调函数。putCacheCallback() 方法中调用的 responseCallback() 回调函数的主要逻辑就是创建 OffsetCommitResponse 并添加到 RequestChannel 中等待发送。

OffsetFetchRequest 与 ListGroupsRequest 相关处理

当 Consumer Group 宕机后重新上线时, 可以通过向 GroupCoordinator 发送 OffsetFetchRequest 获取其最近一次提交的 offset, 并从此位置重新开始进行消费。GroupCoordinator 在收到 OffsetFetchRequest 后会交给 GroupMetadataManager 进行处理, 它会根据请求中的 groupId 查找对应的 OffsetAndMetadata 对象, 并返回给消费者。

整个 OffsetFetchRequest 处理过程比较简单, 我们从 KafkaApis.handleOffsetFetchRequest()

方法开始分析。

```
def handleOffsetFetchRequest(request: RequestChannel.Request) {
    // 省略验证部分的代码
    // 根据 OffsetFetchRequest 的版本号来进行不同的读取 offset 的流程处理。apiVersion
    // 为 0 表示使用旧版本请求，此时的 offset 存储在 ZooKeeper 中，所以其处理逻辑主要是
    // ZooKeeper 的读取操作。在 OffsetCommitRequest 的处理过程中也有根据版本号进行不同
    // 操作的相关代码，请读者注意
    // apiVersion 为 1 表示使用新版本的请求，此时提交的 offset 由 GroupCoordinator 管理
    if (header.apiVersion == 0) {
        ... .. // 旧版本请求的处理省略
    } else {
        // 将 OffsetFetchRequest 委托给 GroupCoordinator 处理
        val offsets = coordinator.handleFetchOffsets(offsetFetchRequest.
groupId,
        authorizedTopicPartitions).toMap
        new OffsetFetchResponse((offsets ++ unauthorizedStatus).asJava)
    }
    // 将 OffsetFetchResponse 放入 RequestChannel 中等待发送
    requestChannel.sendResponse(new Response(request,
        new ResponseSend(request.connectionId, responseHeader, offsetFetchRespo
nse)))
}
```

GroupCoordinator.handleFetchOffsets() 方法在获取 OffsetMetadat 对象之前，会进行一系列的检测，保证 GroupMetadataManager 处于可用状态且是对应 Consumer Group 的管理者。经过验证后，GroupMetadataManager 按照请求指定的分区查找 offset 信息。

```
def handleFetchOffsets(groupId: String, partitions: Seq[TopicPartition]):
    Map[TopicPartition, OffsetFetchResponse.PartitionData] = {
    // 检测 GroupCoordinator 是否是 Consumer Group 的管理者
    if (!isCoordinatorForGroup(groupId)) {
        // 设置错误码（略）
    } else if (isCoordinatorLoadingInProgress(groupId)) {
        // 检测 GroupMetadata 是否已完成加载
        // 设置错误码（略）
    } else {
        // 交给 GroupMetadataManager 处理
        groupManager.getOffsets(groupId, partitions)
    }
```



```

    }
}

// 下面是 GroupMetadataManager.getOffsets() 方法的实现
def getOffsets(group: String, topicPartitions: Seq[TopicPartition]):
    Map[TopicPartition, OffsetFetchResponse.PartitionData] = {
    // 检测 GroupCoordinator 是否是 Consumer Group 的管理者
    if (isGroupLocal(group)) {
        if (topicPartitions.isEmpty) {
            // 如果请求的分区为空, 则表示请求全部分区对应的最近提交的 offset
            offsetsCache.filter(_._1.group == group).map {
                case (groupTopicPartition, offsetAndMetadata) =>
                    (groupTopicPartition.topicPartition, new OffsetFetchResponse.
PartitionData(
                        offsetAndMetadata.offset, offsetAndMetadata.metadata, Errors.
NONE.code))
            }.toMap
        } else {
            // 查找指定分区集合的最近提交 offset
            topicPartitions.map { topicPartition =>
                val groupTopicPartition = GroupTopicPartition(group, topicPartition)
                (groupTopicPartition.topicPartition, getOffset(groupTopicPartition))
            }.toMap
        }
    } else {
        // 异常处理, 返回相应的错误码 (略)
    }
}

```

ListGroupsRequest 一般不会通过生产者或消费者发送, 而是通过管理工具产生, 例如 kafka-consumer-groups 脚本, 该脚本将在第 5 章中介绍。ListGroupsRequest 的处理非常简单, 直接将 GroupMetadataManager 中 groupsCache 集合的数据返回即可, 请读者参考源码学习。

4.7.2 GroupCoordinator 分析

通过上一小节的介绍, 我们了解了 GroupMetadataManager 的原理和实现, 分析了 GroupMetadataManager 如何通过 Offsets Topic 实现 GroupMetadata 信息和 offset 信息的记录工作, 以及 GroupCoordinator 迁移、相关请求处理的实现。

本节开始分析 GroupCoordinator 的实现。首先来看 GroupCoordinator 依赖的组件，如图 4-72 所示。

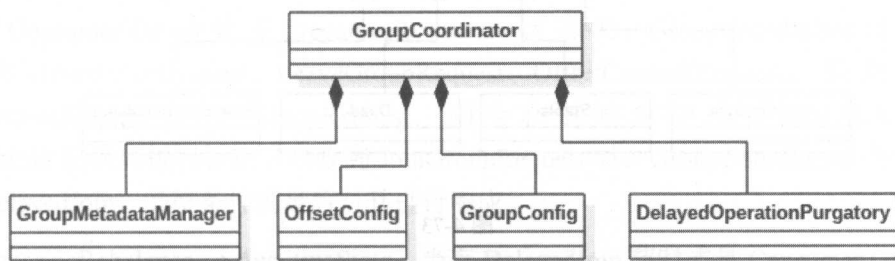


图 4-72

GroupCoordinator 中各个字段的含义和功能如下所述。

- groupConfig: GroupConfig 类型，记录了 Consumer Group 中 Consumer Session 过期的最小时长和最大时长，这个区间是消费者指定的超时时长的合法区间。
- offsetConfig: OffsetConfig 对象，记录了 OffsetMetadata 相关的配置项。例如：OffsetMetadata 中 metadata 字段允许的最大长度、Offsets Topic 中每个分区的副本个数等。
- groupManager: GroupMetadataManager 对象，上一小节已经介绍过，不再赘述。
- heartbeatPurgatory: DelayedOperationPurgatory[DelayedHeartbeat] 类型，用于管理 DelayedHeartbeat。
- joinPurgatory: DelayedOperationPurgatory[DelayedJoin] 对象，用于管理 DelayedJoin。

GroupState

GroupState 接口用于表示 Consumer Group 的状态，它的四个子类分别代表了 Consumer Group 所处的四种不同状态，如图 4-73 所示。该状态是在服务端 GroupCoordinator 中管理 Consumer Group 时使用的状态，并非客户端 Consumer Group 的状态，读者注意区分。

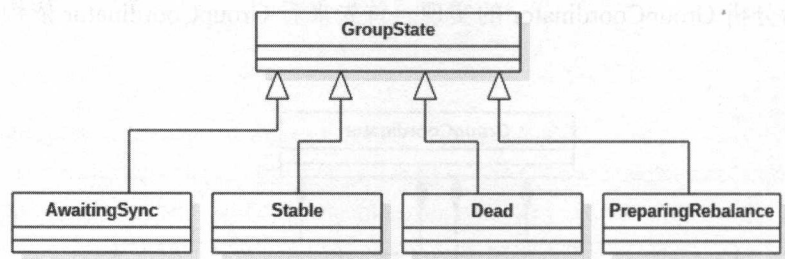


图 4-73

GroupCoordinator 为其管理的每个 Consumer Group 都维护了一个状态机，这四个状态的含义如表 4-8 所示。各个 GroupState 之间的转换如图 4-74 所示。

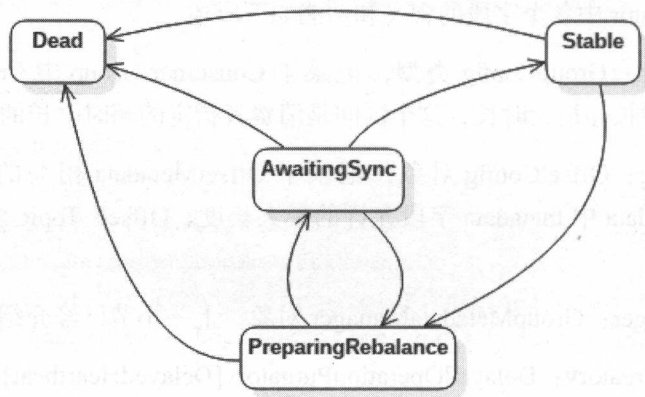


图 4-74

表 4-8

状 态	含 义
PreparingRebalance	Consumer Group 当前正在准备进行 Rebalance 操作
AwaitingSync	Consumer Group 当前正在等待 Group Leader 将分区的分配结果发送到 GroupCoordinator
Stable	标识 Consumer Group 处于正常状态，这也是 Consumer Group 的初始状态
Dead	处于此状态的 Consumer Group 中已经没有 Member 存在了

下面来看一下 GroupCoordinator 针对各个状态的 Consumer Group 能够处理的请求以

及各个状态之间转换的时机。

- PreparingRebalance 状态

当 Consumer Group 处于 PreparingRebalance 状态时, GroupCoordinator 可以正常地处理 OffsetFetchRequest、LeaveGroupRequest、OffsetCommitRequest, 但对于收到的 HeartbeatRequest 和 SyncGroupRequest, 则会在其响应中携带 REBALANCE_IN_PROGRESS 错误码进行标识。当收到 JoinGroupRequest 时, GroupCoordinator 会先创建对应的 DelayedJoin, 等待条件满足后对其进行响应。

PreparingRebalance → AwaitingSync: 当有 DelayedJoin 超时或是 Consumer Group 之前的 Member 都已经重新申请加入时进行切换。

PreparingRebalance → Dead: 所有的 Member 都离开 Consumer Group 时进行切换。

- AwaitingSync 状态

当 Consumer Group 处于 AwaitingSync 状态时, 表示正在等待 Group Leader 的 SyncGroupRequest。当 GroupCoordinator 收到 OffsetCommitRequest 和 HeartbeatRequest 请求时, 会在其响应中携带 REBALANCE_IN_PROGRESS 错误码进行标识。对于来自 Group Follower 的 SyncGroupRequest, 则直接抛弃, 直到收到 Group Leader 的 SyncGroupRequest 时一起响应。

AwaitingSync → Stable: 当 GroupCoordinator 收到 Group Leader 发来的 SyncGroupRequest 时进行切换。

AwaitingSync → PreparingRebalance: 有三种情况可能导致此状态切换, 一是有 Member 加入或退出 Consumer Group, 二是有新的 Member 请求加入 Consumer Group, 三是 Consumer Group 中的 Member 心跳超时。

- Stable 状态

针对该状态的 Consumer Group, GroupCoordinator 可以处理所有的请求, 例如: OffsetFetchRequest、HeartbeatRequest、OffsetCommitRequest、来自 Group Follower 的 JoinGroupRequest、来自 Consumer Group 中现有 Member 的 SyncGroupRequest。

Stable → PreparingRebalance: 有四种情况会导致此状态切换, 一是 Consumer Group 中现有 Member 心跳检测超时, 二是现有 Member 主动退出, 三是当前的 Group Leader 发送 JoinGroupRequest, 四是有新的 Member 请求加入 Consumer Group。

- Dead 状态

处于此状态的 Consumer Group 中没有 Member，其对应的 GroupMetadata 也将被删除。对于此状态的 Consumer Group，除了 OffsetCommitRequest，其他请求的响应中都会携带 UNKNOWN_MEMBER_ID 错误码进行标识。

JoinGroupRequest 分析

通过前面对 GroupCoordinatorRequest 相关处理的介绍，我们了解了 Consumer Group 与 GroupCoordinator 之间的对应关系，这也是 Rebalance 的第一个步骤。Rebalance 过程的第二个步骤是消费者向 GroupCoordinator 发送 JoinGroupRequest，本节重点分析 GroupCoordinator 如何处理 JoinGroupRequest。

JoinGroupRequest 在 API 层是由 KafkaApis.handleJoinGroupRequest() 方法负责处理的，它首先会进行权限验证，之后将 JoinGroupRequest 委托给 GroupCoordinator 进行处理，一同传递给 GroupCoordinator 的还有其中定义的 sendResponseCallback() 回调函数。

```
def handleJoinGroupRequest(request: RequestChannel.Request) {
  // 解析 JoinGroupRequest
  val joinGroupRequest = request.body.asInstanceOf[JoinGroupRequest]
  val responseHeader = new ResponseHeader(request.header.correlationId)

  // sendResponseCallback() 回调函数的定义
  def sendResponseCallback(joinResult: JoinGroupResult) {
    ... ..
    // 创建 JoinGroupResponse
    val responseBody = new JoinGroupResponse(joinResult.errorCode,
      joinResult.generationId, joinResult.subProtocol,
      joinResult.memberId, joinResult.leaderId, members)
    // 将 JoinGroupResponse 放入 RequestChannel 中等待发送
    requestChannel.sendResponse(new RequestChannel.Response(request,
      new ResponseSend(request.connectionId, responseHeader, responseBody)))
  }
  // 进行权限验证
  if (!authorize(request.session, Read, new Resource(Group, joinGroupRequest.groupId())))
  {
    ... .. // 验证失败的处理 (略)
  } else {
```

```

// 将 JoinGroupRequest 交给 GroupCoordinator.handleJoinGroup() 方法进行处理
coordinator.handleJoinGroup(joinGroupRequest.groupId, joinGroupRequest.
memberId,
    request.header.clientId, request.session.clientAddress.toString,
    joinGroupRequest.sessionTimeout, joinGroupRequest.protocolType,
    protocols, sendResponseCallback)
}
}

```

在 `GroupCoordinator.handleJoinGroup()` 方法中，首先会进行一系列的检测，保证 `GroupCoordinator` 处于可用状态且是对应 `Consumer Group` 的管理者。还要对 `sessionTimeoutMs` 进行检查，它是由消费者设置的超时时长，即消费者发送 `HeartbeatRequest` 的最长时间间隔，它不能超出 `GroupConfig` 中配置的超时时长区间。之后检测 `memberId` 的合法性，根据 `groupId` 决定是否创建 `GroupMetadata` 对象。最后调用 `doJoinGroup()` 方法继续后续处理。

```

def handleJoinGroup(groupId: String, memberId: String, clientId: String,
    clientHost: String, sessionTimeoutMs: Int, protocolType: String,
    protocols:
        List[(String, Array[Byte])], responseCallback: JoinCallback) {
    if (!isActive.get) { // 检测 GroupCoordinator 是否启动
        ..... // 返回错误码 (略)
    } else if (!validGroupId(groupId)) { // 检测 groupId 是否合法
        ..... // 返回错误码 (略)
    } else if (!isCoordinatorForGroup(groupId)) {
        // 检测 GroupCoordinator 是否管理此 Consumer Group
        ..... // 返回错误码 (略)
    } else if (isCoordinatorLoadingInProgress(groupId)) {
        // GroupCoordinator 是否已经加载此 Consumer Group 对应的 Offsets Topic 分区
        // 返回错误码 (略)
    } else if (sessionTimeoutMs < groupConfig.groupMinSessionTimeoutMs
        || sessionTimeoutMs > groupConfig.groupMaxSessionTimeoutMs) {
        // 检测 Consumer 指定超时时长是否在合法区间
        ..... // 返回错误码 (略)
    } else {
        var group = groupManager.getGroup(groupId)
        if (group == null) {

```

```

        // 检测 memberId 是否是合法的
        if (memberId != JoinGroupRequest.UNKNOWN_MEMBER_ID) {
            ..... // 返回错误码 (略)
        } else {
            // 创建 GroupMetadata 对象
            group = groupManager.addGroup(new GroupMetadata(groupId,
            protocolType))
            // 调用 doJoinGroup() 方法完成后续功能
            doJoinGroup(group, memberId, clientId, clientHost,
            sessionTimeoutMs,
            protocolType, protocols, responseCallback)
        }
    } else {
        doJoinGroup(group, memberId, clientId, clientHost, sessionTimeoutMs,
        protocolType, protocols, responseCallback)
    }
}
}
}

```

GroupCoordinator.doJoinGroup() 方法首先还会做两方面的检测：一方面是检测 Member 支持的 PartitionAssignor，这里需要检测每个消费者支持的 PartitionAssignor 集合与 GroupMetadata 中的候选 PartitionAssignor 集合（即 candidateProtocols 字段）是否有交集，只有这样才能选择出所有 Consumer 都支持的 PartitionAssignor；另一方面是检测 memberId，JoinGroupRequest 可能是来自 Consumer Group 中已知的 Member，此时请求会携带之前被分配过的 memberId，这里就要检测 memberId 是否能被 GroupMetadata 识别。之后，按照当前 Consumer Group 所处的状态分类处理：

- Dead

直接返回 UNKNOWN_MEMBER_ID 错误码。

- PreparingRebalance

a) 如果是已知 Member 重新申请加入，则更新 GroupMetadata 中记录的 Member 信息。

b) 如果是未知的新 Member 申请加入，则创建 Member 并分配 memberId，并加入 GroupMetadata 中。

这里要注意，Member.awaitingJoinCallback 不仅是一个回调函数，还是一个非常重要的标识，它会参与检测所有已知 Member 是否已经发送 JoinGroupRequest 申请重新加入。

- AwaitingSync

如果是未知的新 Member 申请加入，则创建 Member 并分配 memberId，并加入 GroupMetadata 中。然后调用 maybePrepareRebalance() 方法将状态切换为 PreparingRebalance，其中涉及的具体操作下面详述。

如果是已知 Member 重新申请加入，则要区分 Member 支持的 PartitionAssignor 是否发生了变化：若未发生变化，则将当前 Member 集合信息返回给 Group Leader。若发生变化，则更新 Member 信息，并调用 maybePrepareRebalance() 方法将状态切换为 PreparingRebalance。

- Stable

a) 如果是未知的新 Member 申请加入，则创建 Member 并分配 memberId，并加入 GroupMetadata 中。然后调用 maybePrepareRebalance() 方法将状态切换为 PreparingRebalance，其中涉及的具体操作下面详述。

b) 如果是已知 Member 重新申请加入，则要区分 Member 支持的 PartitionAssignor 是否发生了变化：如果发生变化或者发送 JoinGroupRequest 请求的是 Group Leader，则更新 Member 信息，并调用 maybePrepareRebalance() 方法将状态切换为 PreparingRebalance。如果未发生变化则将 GroupMetadata 的当前状态返回，消费者会发送 SyncGroupRequest 继续后面的操作。

最后，根据当前的状态决定是否尝试完成相关的 DelayedJoin 操作。

GroupCoordinator.doJoinGroup() 方法的具体实现如下：

```
private def doJoinGroup(group: GroupMetadata, memberId: String, clientId:
String,
                                clientHost: String, sessionTimeoutMs: Int,
protocolType: String,
                                protocols: List[(String, Array[Byte])],
responseCallback: JoinCallback) {
  group synchronized {
    // 检测 Member 支持的 PartitionAssignor
    if (group.protocolType != protocolType ||
        !group.supportsProtocols(protocols.map(_._1).toSet)) {
      ..... // 返回错误码 (略)
    } else if (memberId != JoinGroupRequest.UNKNOWN_MEMBER_ID
        && !group.has(memberId)) { // 检测 memberId 是否能被识别
```



```

... .. // 返回错误码 (略)
} else {
    // 根据 Consumer Group 的状态分类进行处理
    group.currentState match {
        case Dead => ... .. // 返回错误码 (略)
        case PreparingRebalance =>
            // 根据 memberId 是否为 UNKNOWN_MEMBER_ID 判断消费者是否为已知 Member
            // 未知 Member 申请加入
            if (memberId == JoinGroupRequest.UNKNOWN_MEMBER_ID) {
                addMemberAndRebalance(sessionTimeoutMs, clientId, clientHost,
                    protocols, group, responseCallback)
            } else { // 已知 Member 重新申请加入
                val member = group.get(memberId)
                updateMemberAndRebalance(group, member, protocols,
                    responseCallback)
            }
        case AwaitingSync =>
            if (memberId == JoinGroupRequest.UNKNOWN_MEMBER_ID) {
                // 未知 Member 申请加入会发生状态切换
                addMemberAndRebalance(sessionTimeoutMs, clientId, clientHost,
                    protocols,
                        group, responseCallback)
            } else {
                // 已知 Member 重新申请加入
                val member = group.get(memberId)
                if (member.matches(protocols)) {
                    // 支持的 PartitionAssignor 未发生改变, 返回 GroupMetadata 的信息
                    responseCallback(JoinGroupResult(
                        members = if (memberId == group.leaderId) {
                            group.currentMemberMetadata
                        } else {
                            Map.empty
                        }, memberId = memberId,
                        generationId = group.generationId, subProtocol = group.
                            protocol,
                        leaderId = group.leaderId, errorCode = Errors.NONE.code))
                } else {

```

```

        // 支持的 PartitionAssignor 发生改变, 需要更新 Member 信息并发生状态切换
        updateMemberAndRebalance(group, member, protocols,
responseCallback)
    }
}
case Stable =>
    if (memberId == JoinGroupRequest.UNKNOWN_MEMBER_ID) {
        // 未知 Member 申请加入会发生状态切换
        addMemberAndRebalance(sessionTimeoutMs, clientId, clientHost,
protocols,
        group, responseCallback)
    } else {
        // 已知 Member 重新申请加入
        val member = group.get(memberId)
        if (memberId == group.leaderId || !member.matches(protocols)) {
            // Group Leader 或支持的 PartitionAssignor 发生改变,
            // 则更新 Member 信息并发生状态切换
            updateMemberAndRebalance(group, member, protocols,
responseCallback)
        } else {
            // 支持的 PartitionAssignor 未发生改变, 返回 GroupMetadata 的信息
            responseCallback(JoinGroupResult(members = Map.empty,
memberId = memberId,
            generationId = group.generationId, subProtocol = group.
protocol,
            leaderId = group.leaderId, errorCode = Errors.NONE.code))
        }
    }
}
if (group.is(PreparingRebalance)) // 尝试完成相关的 DelayedJoin
    joinPurgatory.checkAndComplete(GroupKey(group.groupId))
}
}
}

```

在 `GroupCoordinator.doJoinGroup()` 方法中多次出现 `addMemberAndRebalance()` 和 `updateMemberAndRebalance()` 这两个方法, 它们负责了添加 / 更新 Member 信息, 并调用 `prepareRebalance()` 方法实现 GroupMetadata 的状态切换。该方法调用栈如图 4-75 所示。

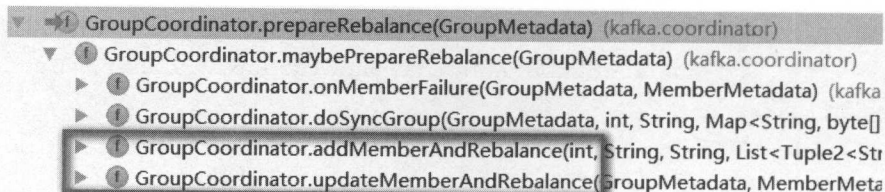


图 4-75

GroupCoordinator.addMemberAndRebalance() 方法的实现如下：

```
private def addMemberAndRebalance(sessionTimeoutMs: Int, clientId: String,
    clientHost: String, protocols: List[(String, Array[Byte])],
    group: GroupMetadata, callback: JoinCallback) = {
  // memberId 由 clientId 与 UUID 构成
  val memberId = clientId + "-" + group.generateMemberIdSuffix
  // 创建新的 MemberMetadata 对象
  val member = new MemberMetadata(memberId, group.groupId, clientId,
    clientHost, sessionTimeoutMs, protocols)
  // 设置 awaitingJoinCallback, 该回调函数是 KafkaApis.handleJoinGroupRequest()
  // 方法中定义的 sendResponseCallback() 方法回调函数
  member.awaitingJoinCallback = callback
  group.add(member.memberId, member) // 添加到 GroupMetadata 中保存
  maybePrepareRebalance(group) // 尝试进行状态切换
  member
}
```

maybePrepareRebalance() 方法会判断 Consumer Group 的状态，如果是 Stable 或 AwaitingSync，则会调用 prepareRebalance() 方法将状态切换成 PreparingRebalance，并创建相应的 DelayedJoin。

```
private def prepareRebalance(group: GroupMetadata) {
  // 如果处于 AwaitingSync 状态，则先要重置 MemberMetadata.assignment 字段，
  // 并回调 awaitingSyncCallback 向消费者返回 REBALANCE_IN_PROGRESS 的错误码
  if (group.is(AwaitingSync))
    resetAndPropagateAssignmentError(group, Errors.REBALANCE_IN_PROGRESS.
code)
  // 将 Consumer Group 状态转换为 PreparingRebalance 状态，
  // 表示准备执行 Rebalance 的操作
  group.transitionTo(PreparingRebalance) // 切换为 PreparingRebalance 状态
```



```

// DelayedJoin 的超时时长是 GroupMetadata 中所有 Member 设置的超时时长的最大值
val rebalanceTimeout = group.rebalanceTimeout
// 创建 DelayedJoin 对象
val delayedRebalance = new DelayedJoin(this, group, rebalanceTimeout)
val groupKey = GroupKey(group.groupId) // 创建 DelayedJoin 的 Key

// 尝试立即完成 DelayedJoin, 否则将 DelayedFetch 添加到 joinPurgatory 中
joinPurgatory.tryCompleteElseWatch(delayedRebalance, Seq(groupKey))
}

```

对于 AwaitingSync 状态的 Consumer Group 来说, 有的 Group Follower 已经发送了 SyncGroupRequest, GroupCoordinator 在等待 Group Leader 通过 SyncGroupRequest 将分区的分配结果发送过来。如果此时进行状态切换, 需要对这些已经发送 SyncGroupRequest 的 Group Follower 返回错误码, 这部分操作在 resetAndPropagateAssignmentError() 方法中完成。

```

private def resetAndPropagateAssignmentError(group: GroupMetadata,
errorcode: Short) {
    assert(group.is(AwaitingSync))
    // 清空所有 MemberMetadata 的 assignment 字段
    group.allMemberMetadata.foreach(_._assignment = Array.empty[Byte])
    propagateAssignment(group, errorcode)
}

// 下面是 propagateAssignment() 方法的实现
private def propagateAssignment(group: GroupMetadata, errorcode: Short) {
    for (member <- group.allMemberMetadata) {
        if (member.awaitingSyncCallback != null) {
            // 调用 awaitingSyncCallback 回调函数, 向对应的 Consumer 发送 SyncGroupResponse
            member.awaitingSyncCallback(member._assignment, errorcode)
            member.awaitingSyncCallback = null // 清空 awaitingSyncCallback 回调函数
            // 开启等待下次心跳的延迟任务, 心跳消息的相关内容后面详述
            completeAndScheduleNextHeartbeatExpiration(group, member)
        }
    }
}
}

```

现在再来分析 GroupCoordinator.updateMemberAndRebalance() 方法就没有什么难度了,

会发现与 `addMemberAndRebalance()` 方法非常类似:

```
private def updateMemberAndRebalance(group: GroupMetadata, member:
MemberMetadata,
    protocols: List[(String, Array[Byte])], callback: JoinCallback) {
    // 更新 MemberMetadata 支持的协议和 awaitingJoinCallback 回调函数
    member.supportedProtocols = protocols
    member.awaitingJoinCallback = callback
    maybePrepareRebalance(group) // 尝试进行状态切换
}
```

DelayedJoin 分析

通过前面对 `JoinGroupRequest` 处理流程的分析, 我们知道到 `Consumer Group` 的状态切换成 `PreparingRebalance` 时会创建一个 `DelayedJoin` 对象并添加到 `GroupCoordinator.joinPurgatory` 中管理。前面我们已经介绍了 `DelayedOperationPurgatory` 的相关内容, 也分析了 `DelayedProduce` 和 `DelayedFetch` 的具体实现。`DelayedJoin` 与它们类似, 也是一种延迟操作, 它的主要功能是等待 `Consumer Group` 中所有的消费者发送 `JoinGroupRequest` 申请加入。每当处理完新收到的 `JoinGroupRequest` 时, 都会检测相关的 `DelayedJoin` 是否能够完成, 经过一段时间的等待, `DelayedJoin` 也会到期执行。

`DelayedJoin` 中各个字段的含义如下所述。

- `Coordinator`: `GroupCoordinator` 对象, `DelayedJoin` 中方法的实现方式是调用 `GroupCoordinator` 中对应的方法。
- `Group`: `DelayedJoin` 对应的 `GroupMetadata` 对象。
- `sessionTimeout`: 指定了 `DelayedJoin` 的到期时长, 此时间是 `GroupMetadata` 中所有 `Member` 设置的超时时间的最大值。

`DelayedJoin` 实现了 `DelayedOperation` 这个抽象类, 其具体实现如下:

```
override def tryComplete(): Boolean =
    coordinator.tryCompleteJoin(group, forceComplete)

override def onExpiration() = coordinator.onExpireJoin() // 空实现

override def onComplete() = coordinator.onCompleteJoin(group)
```

在 `GroupCoordinator.tryCompleteJoin()` 方法中, 通过判断 `GroupMetadata` 中的已知

Member 是否已重新申请加入来决定 DelayedJoin 是否符合执行条件。

```
def tryCompleteJoin(group: GroupMetadata, forceComplete: () => Boolean) = {
  group synchronized {
    // 判断已知 Member 是否已经申请加入
    if (group.notYetRejoinedMembers.isEmpty) {
      forceComplete()
    }
    else {
      false
    }
  }
}

// 前面介绍过 awaitingJoinCallback 是个非常重要的标记，它用来判断一个 Member 是否已经
// 申请加入
def notYetRejoinedMembers = members.values
  .filter(_.awaitingJoinCallback == null).toList
```

当已知 Member 都已申请重新加入或 DelayedJoin 到期时，都会执行 GroupCoordinator.onComplete() 方法。在该方法中首先将未重新申请加入的已知 Member 删除，如果 GroupMetadata 中不再包含任何 Member，则将 Consumer Group 转换成 Dead 状态，删除对应的 GroupMetadata 对象并向 Offsets Topic 中对应的分区写入“删除标记”，之后更新 generationId，调用 awaitingJoinCallback() 回调函数发送 JoinGroupResponse。

```
def onCompleteJoin(group: GroupMetadata) {
  group synchronized {
    // 获取未重新申请加入的已知 Member 的集合
    val failedMembers = group.notYetRejoinedMembers
    if (group.isEmpty || !failedMembers.isEmpty) {
      failedMembers.foreach { failedMember =>
        group.remove(failedMember.memberId) // 移除未加入的已知 Member
      }
      // 如果 GroupMetadata 中已经没有 Member，则将 GroupMetadata 切换成 Dead 状态
      if (group.isEmpty) {
        group.transitionTo(Dead)
      }
    }
  }
}
```

```

        // 删除GroupMetadata对象,并在对应的Offsets Topic Partition中添加“删除标记”
        groupManager.removeGroup(group)
    }
}

if (!group.is(Dead)) {
    // 在GroupMetadata.initNextGeneration()方法中不仅 generationId 递增,还会选择
    // 该 Consumer Group 最终使用的 PartitionAssignor
    group.initNextGeneration()
    // 向 GroupMetadata 中所有的 Member 发送 JoinGroupResponse
    for (member <- group.allMemberMetadata) {
        val joinResult = JoinGroupResult(
            // 注意,发送给 Group Leader 和 Group Follower 的 JoinGroupResponse 内容不同
            members = if (member.memberId == group.leaderId) {
                group.currentMemberMetadata
            } else {
                Map.empty
            }, memberId = member.memberId, generationId = group.generationId,
            subProtocol = group.protocol, leaderId = group.leaderId,
            errorCode = Errors.NONE.code)

        // 调用回调函数,即 KafkaApis.handleJoinGroupRequest() 方法中定义的
        // sendResponseCallback() 方法,将响应放入 RequestChannel 中,等待发送
        member.awaitingJoinCallback(joinResult)
        member.awaitingJoinCallback = null
        completeAndScheduleNextHeartbeatExpiration(group, member) // 心跳操作
    }
}
}
}
}

```

HeartbeatRequest 分析

在第3章中介绍过,每个消费者都会通过 HeartbeatTask 任务定期向 GroupCoordinator 发送 HeartbeatRequest,告知 GroupCoordinator 自己正常在线。

HeartbeatRequest 首先由 KafkaApis.handleHeartbeatRequest() 方法进行处理,它负责验证权限,定义回调函数,并将请求委托给 GroupCoordinator 处理。


```

def handleHeartbeatRequest(request: RequestChannel.Request) {
    val heartbeatRequest = request.body.asInstanceOf[HeartbeatRequest]
    val respHeader = new ResponseHeader(request.header.correlationId)

    // 定义回调函数，将 HeartbeatResponse 放入 RequestChannel 等待发送
    def sendResponseCallback(errorCode: Short) {
        val response = new HeartbeatResponse(errorCode)
        requestChannel.sendResponse(new RequestChannel.Response(request,
            new ResponseSend(request.connectionId, respHeader, response)))
    }

    // 省略权限验证的相关代码
    // 将 HeartbeatRequest 委托给 GroupCoordinator 处理
    coordinator.handleHeartbeat(heartbeatRequest.groupId(),
        heartbeatRequest.memberId(), heartbeatRequest.groupGenerationId(),
        sendResponseCallback)
}

```

`GroupCoordinator.handleHeartbeat()` 方法首先会进行一系列的检测，保证 `GroupMetadataManager` 处于可用状态且是对应 `Consumer Group` 的管理者。之后检测 `Consumer Group` 状态、`MemberId`、`generationId` 是否合法。最后，调用 `completeAndScheduleNext-HeartbeatExpiration()` 方法，继续下一步工作。

```

def handleHeartbeat(groupId: String, memberId: String, generationId: Int,
    responseCallback: Short => Unit) {
    if (!isActive.get) {
        responseCallback(Errors.GROUP_COORDINATOR_NOT_AVAILABLE.code)
    } else if (!isCoordinatorForGroup(groupId)) {
        // 检测 GroupCoordinator 是否管理该 Consumer Group
        responseCallback(Errors.NOT_COORDINATOR_FOR_GROUP.code)
    } else if (isCoordinatorLoadingInProgress(groupId)) {
        responseCallback(Errors.NONE.code) // 是否已经加载对应的 Offsets Topic 分区
    } else {
        val group = groupManager.getGroup(groupId)
        if (group == null) { // 检测 GroupMetadata 是否存在
            responseCallback(Errors.UNKNOWN_MEMBER_ID.code)
        } else {
            group.synchronized {
                if (group.is(Dead)) { // 检测 Consumer Group 的状态

```



```

        responseCallback(Errors.UNKNOWN_MEMBER_ID.code)
    } else if (!group.is(Stable)) {
        responseCallback(Errors.REBALANCE_IN_PROGRESS.code)
    } else if (!group.has(memberId)) { // 检测 MemberId
        responseCallback(Errors.UNKNOWN_MEMBER_ID.code)
    } else if (generationId != group.generationId) { // 检测 generationId
        responseCallback(Errors.ILLEGAL_GENERATION.code)
    } else {
        val member = group.get(memberId)
        // 通过 completeAndScheduleNextHeartbeatExpiration() 继续下一步操作
        completeAndScheduleNextHeartbeatExpiration(group, member)
        responseCallback(Errors.NONE.code)
    }
}
}
}
}
}

```

在 `completeAndScheduleNextHeartbeatExpiration()` 方法中会更新收到此 Member 心跳的时间戳，尝试执行其对应的 `DelayedHeartbeat`，并创建新的 `DelayedHeartbeat` 对象放入 `heartbeatPurgatory` 中等待下次心跳到来或 `DelayedHeartbeat` 超时。

```

private def completeAndScheduleNextHeartbeatExpiration(group: GroupMetadata,
    member: MemberMetadata) {
    member.latestHeartbeat = time.milliseconds() // 更新心跳时间
    // 获取 DelayedHeartbeat 的 Key
    val memberKey = MemberKey(member.groupId, member.memberId)
    // 尝试完成之前添加的 DelayedHeartbeat
    heartbeatPurgatory.checkAndComplete(memberKey)
    // 计算下一次的 Heartbeat 的超时时间
    val newHeartbeatDeadline = member.latestHeartbeat + member.
sessionTimeoutMs
    // 创建新的 DelayedHeartbeat 对象，并添加到 heartbeatPurgatory 中管理
    val delayedHeartbeat = new DelayedHeartbeat(this,
        group, member, newHeartbeatDeadline, member.sessionTimeoutMs)
    heartbeatPurgatory.tryCompleteElseWatch(delayedHeartbeat, Seq(memberKey))
}

```

在前面分析过程中多次看到 `completeAndScheduleNextHeartbeatExpiration()` 方法的身影,其调用栈如图 4-76 所示。了解 `completeAndScheduleNextHeartbeatExpiration()` 方法的功能后,就可以明白为什么 `GroupCoordinator` 会在这些位置对其调用,因为 `GroupCoordinator` 也会将 `OffsetCommitRequest` 等请求当作消费者在线的标志,并重置收到心跳的时间戳。



图 4-76

介绍完 `HeartbeatRequest` 的处理过程,再来分析 `DelayedHeartbeat` 的实现。`DelayedHeartbeat` 中各字段的含义如下所述。

- `group`: 对应的 `GroupMetadata` 对象。
- `member`: 对应的 `MemberMetadata` 对象。
- `heartbeatDeadline`: `DelayedHeartbeat` 的到期时间戳。
- `sessionTimeout`: 指定了 `DelayedHeartbeat` 的到期时长,此时间是消费者在 `JoinGroupRequest` 中设置的,且符合 `GroupConfig` 指定的合法区间。
- `coordinator`: `GroupCoordinator` 对象, `DelayedHeartbeat` 中方法的实现方式是调用 `GroupCoordinator` 中对应的方法。

`DelayedHeartbeat` 实现了 `DelayedOperation` 这个抽象类,具体实现如下:

```
override def tryComplete(): Boolean =
  coordinator.tryCompleteHeartbeat(group, member, heartbeatDeadline,
    forceComplete)

override def onExpiration() =
  coordinator.onExpireHeartbeat(group, member, heartbeatDeadline)

override def onComplete() = coordinator.onCompleteHeartbeat() // 空实现
```

在 `GroupCoordinator.tryCompleteHeartbeat()` 方法中会检测下列四个条件,如果满足其中的任意一个条件,则认为 `DelayedHeartbeat` 符合执行条件。

- (1) 最后一次收到心跳信息的时间与 heartbeatDeadline 的差距大于 sessionTimeout。
- (2) awaitingJoinCallback 不为 null，即消费者正在等待 JoinGroupResponse。
- (3) awaitingSyncCallback 不为 null，即消费者正在等待 SyncGroupResponse。
- (4) 消费者已经离开了 Consumer Group。

tryCompleteHeartbeat() 方法的具体代码实现如下：

```
def tryCompleteHeartbeat(group: GroupMetadata, member: MemberMetadata,
    heartbeatDeadline: Long, forceComplete: () => Boolean) = {
  group synchronized {
    if (shouldKeepMemberAlive(member, heartbeatDeadline) // 检测条件 1~3
        || member.isLeaving) { // 检测条件 4
      forceComplete()
    } else false
  }
}

// 下面是 shouldKeepMemberAlive() 方法的代码
private def shouldKeepMemberAlive(member: MemberMetadata,
    heartbeatDeadline: Long) =
  member.awaitingJoinCallback != null || // 检测条件 2
  member.awaitingSyncCallback != null || // 检测条件 3
  member.latestHeartbeat + member.sessionTimeoutMs > heartbeatDeadline
// 检测条件 1
```

GroupCoordinator.onCompleteHeartbeat() 方法是空实现，故 DelayedHeartbeat 执行之后仅会将其从 heartbeatPurgatory 中删除，并不会进行其他操作。

DelayedHeartbeat 到期执行时还会调用 GroupCoordinator.onExpireHeartbeat() 方法，它会将其对应的 Member 从 GroupMetadata 中删除，并按照当前 GroupMetadata 所处的状态进行分类处理。

```
def onExpireHeartbeat(group: GroupMetadata, member: MemberMetadata,
    heartbeatDeadline: Long) {
  group synchronized {
    // 再次检测 Member 是否下线
    if (!shouldKeepMemberAlive(member, heartbeatDeadline))
      onMemberFailure(group, member) // Member 下线后的相关处理操作
```



```

    }
}

// 下面是 onMemberFailure() 方法的实现
private def onMemberFailure(group: GroupMetadata, member: MemberMetadata) {
    group.remove(member.memberId) // 将对应的 Member 从 GroupMetadata 中删除
    group.currentState match {
        case Dead => // 不做任何处理

        // 之前的分区分配可能已经失效了, 将 GroupMetadata 切换到 PreparingRebalance 状态
        // maybePrepareRebalance() 方法在前面已经分析过了, 这里不再赘述
        case Stable | AwaitingSync => maybePrepareRebalance(group)
        // GroupMetadata 中的 Member 减少, 可能满足 DelayedJoin 的执行条件, 尝试执行
        case PreparingRebalance => joinPurgatory.checkAndComplete(GroupKey(group.groupId))
    }
}

```

SyncGroupRequest 分析

分析完处理 GroupCoordinatorRequest 和 JoinGroupRequest 的相关实现, 我们继续来分析服务端对 SyncGroupRequest 的处理, 这也是 Rebalance 操作的第三步。API 层由 KafkaApis.handleSyncGroupRequest() 方法负责处理 SyncGroupRequest, 其中会定义相关的回调函数, 并将请求交给 GroupCoordinator.handleSyncGroup() 方法进行处理。

GroupCoordinator.handleSyncGroup() 首先会进行一系列的检测, 保证 GroupCoordinator 处于可用状态且是对应 Consumer Group 的管理者, 具体实现与前面介绍的 GroupCoordinator.handleJoinGroup() 方法类似, 最后将请求交给 GroupCoordinator.doSyncGroup() 方法处理, 调用栈如图 4-77 所示。

```

▼ ① GroupCoordinator.doSyncGroup(GroupMetadata, int, String, Map<String, byte[]>, Function2<byte[], Object, BoxedUnit>)
  ▼ ② GroupCoordinator.handleSyncGroup(String, int, String, Map<String, byte[]>, Function2<byte[], Object, BoxedUnit>) (kaf
    ► ③ KafkaApis.handleSyncGroupRequest(Request) (kafka.server)

```

图 4-77

doSyncGroup() 首先还会做两次检测: 一个是检测 Member 是否为此 Consumer Group 的成员; 二是检测 generationId 是否合法, 这是为了屏蔽来自旧 Member 成员的请求。之后, 按照当前 Consumer Group 所处的状态分类处理:

- Dead

直接返回 UNKNOWN_MEMBER_ID 错误码。

- PreparingRebalance

直接返回 REBALANCE_IN_PROGRESS 错误码。

- AwaitingSync

首先设置 awaitingSyncCallback，更新最后一次收到心跳的时间戳，完成之前相关的 DelayedHeartbeat 并创建新的 DelayedHeartbeat 对象等待下次心跳到来。如果发送请求的是 Group Follower 则直接抛弃；如果发送 SyncGroupRequest 的是 Group Leader，其中会包含分区的分配结果，服务端会通过 GroupMetadataManager 中的 prepareStoreGroup() 方法和 store() 方法将分区的分配结果保存到对应的 Offsets Topic 分区中，并修改 GroupMetadata 中的相关缓存记录，这两个方法的具体实现在前面已经介绍过了，请读者回顾相关小节。

- Stable

将分区的分配结果中与 Member 相关的信息返回。

GroupCoordinator.doSyncGroup() 方法的具体实现如下：

```
private def doSyncGroup(group: GroupMetadata, generationId: Int, memberId:
String,
                        groupAssignment: Map[String, Array[Byte]], responseCallback:
SyncCallback) {
  var delayedGroupStore: Option[DelayedStore] = None
  group synchronized {
    if (!group.has(memberId)) { // 检测 memberId
      responseCallback(Array.empty, Errors.UNKNOWN_MEMBER_ID.code)
    } else if (generationId != group.generationId) { // 检测 generationId
      responseCallback(Array.empty, Errors.ILLEGAL_GENERATION.code)
    } else {
      group.currentState match {
        case Dead => // 直接返回 UNKNOWN_MEMBER_ID 错误码
          responseCallback(Array.empty, Errors.UNKNOWN_MEMBER_ID.code)
        case PreparingRebalance => // 直接返回 REBALANCE_IN_PROGRESS 错误码
          responseCallback(Array.empty, Errors.REBALANCE_IN_PROGRESS.code)

        case AwaitingSync =>
```

```

// 设置 awaitingSyncCallback 回调函数
group.get(memberId).awaitingSyncCallback = responseCallback
// 更新最后一次收到心跳的时间戳，并创建新的 DelayedHeartbeat 对象等待下次
// 心跳到来
    completeAndScheduleNextHeartbeatExpiration(group, group.
get(memberId))
// 处理 GroupLeader 发来的 SyncGroupRequest
if (memberId == group.leaderId) {
// 将未分配到分区的 Member 对应的分配结果填充为空的 Byte 数组
val missing = group.allMembers -- groupAssignment.keySet
    val assignment = groupAssignment ++ missing.map(_ -> Array.
empty[Byte]).toMap
// 通过 GroupMetadataManager 将 GroupMetadata 相关信息形成消息，并写
// 入到对应的 Offsets Topic 分区中
delayedGroupStore = Some(groupManager.prepareStoreGroup(
    group, assignment, (errorCode: Short) => {
        group synchronized {
            if (group.is(AwaitingSync) && generationId == group.
generationId) {
                if (errorCode != Errors.NONE.code) {
                    // 清空分区的分配结果，发送异常响应
                    resetAndPropagateAssignmentError(group, errorCode)
                    // 切换成 PreparingRebalance 状态
                    maybePrepareRebalance(group)
                } else {
                    // 设置分区的分配结果，发送正常的 SyncGroupResponse
                    setAndPropagateAssignment(group, assignment)
                    group.transitionTo(Stable)
                }
            }
        }
    })
})
}

case Stable =>
// 将分配给此 Member 的负责处理的分区信息返回
val memberMetadata = group.get(memberId)
responseCallback(memberMetadata.assignment, Errors.NONE.code)
// 心跳相关操作

```

```

        completeAndScheduleNextHeartbeatExpiration(group, group.
get(memberId))
    }
}
}
delayedGroupStore.foreach(groupManager.store)
}

```

OffsetCommitRequest 分析

在上一节中分析了 GroupMetadataManager 如何将 offset 信息转换成消息并追加到对应的 Offsets Topic 中。本节简单介绍 KafkaApis 和 GroupCoordinator 中对 OffsetCommitRequest 请求的处理。

在 API 层 由 KafkaApis.handleOffsetCommitRequest() 方法负责处理 OffsetCommitRequest，它首先对请求进行权限验证，并过滤掉当前 MetadataCache 中未知的 Topic 对应的 offset 信息。然后根据 OffsetCommitRequest 的版本号决定记录 offset 的消息的超时时长。最后，创建 OffsetAndMetadata 对象，委托给 GroupCoordinator.handleCommitOffsets() 方法进行处理。与前面其他请求的处理类似，在 handleOffsetCommitRequest() 中也定义了用户创建响应并返回的回调函数。

```

def handleOffsetCommitRequest(request: RequestChannel.Request) {
    val header = request.header
    val offsetCommitRequest = request.body.asInstanceOf[OffsetCommitRequest]
    if (!authorize(request.session, Read,
        new Resource(Group, offsetCommitRequest.groupId))) { // 对请求进行权限验证
        // 验证失败，返回响应错误码
        requestChannel.sendResponse(new RequestChannel.Response(request,
            new ResponseSend(request.connectionId, responseHeader, responseBody)))
    } else {
        // 过滤掉当前 MetadataCache 中未知的 Topic 对应的 offset 信息
        val invalidRequestsInfo = offsetCommitRequest.offsetData.asScala.filter
        {
            case (topicPartition, _) => !metadataCache.contains(topicPartition.
topic)
        }
        val filteredRequestInfo = offsetCommitRequest.offsetData.asScala.toMap
    }
}

```



```

invalidRequestsInfo.keys
.....
// 定义回调函数，主要负责创建 OffsetCommitResponse，并放入 RequestChannel 等待发送
def sendResponseCallback(commitStatus: immutable.Map[TopicPartition,
Short]) {
    ..... // 省略一系列日志操作以及集合合并的相关操作
    val responseHeader = new ResponseHeader(header.correlationId)
    val responseBody = new OffsetCommitResponse(combinedCommitStatus.
asJava)
    requestChannel.sendResponse(new RequestChannel.Response(request,
        new ResponseSend(request.connectionId, responseHeader,
responseBody)))
}
..... // 没有可用的 offset 信息，调用 sendResponseCallback() 方法

if (header.apiVersion == 0) {
    // 根据 OffsetCommitRequest 的版本号来进行不同的流程处理。apiVersion 为 0 表示
    // 使用旧版本请求
    // 此时的 offset 信息应该存储在 ZooKeeper 中，所以处理逻辑主要是 ZooKeeper 的写
    // 入操作。这与 OffsetFetchRequest 的处理类似。具体的 ZooKeeper 写入操作略
} else {
    // 根据请求的版本号，决定记录 offset 的消息的超时长
    val offsetRetention = if (header.apiVersion <= 1 ||
        offsetCommitRequest.retentionTime ==
        OffsetCommitRequest.DEFAULT_RETENTION_TIME)
        coordinator.offsetConfig.offsetsRetentionMs // 默认配置 24 小时
    else
        offsetCommitRequest.retentionTime // 请求中指定的时长

    // 根据配置的保留时间，或者每个分区指定的保留时间，计算出 offset 的过期清理的时间
    val currentTimestamp = SystemTime.milliseconds
    val defaultExpireTimestamp = offsetRetention + currentTimestamp
    val partitionData = authorizedRequestInfo.mapValues { partitionData
=>
        val metadata = if (partitionData.metadata == null) OffsetMetadata.
NoMetadata
        else partitionData.metadata
        new OffsetAndMetadata( // 创建 OffsetAndMetadata 对象

```



```

        offsetMetadata = OffsetMetadata(partitionData.offset, metadata),
        commitTimestamp = currentTimestamp,
        expireTimestamp = {
            if (partitionData.timestamp == OffsetCommitRequest.DEFAULT_
TIMESTAMP)
                defaultExpireTimestamp
            else offsetRetention + partitionData.timestamp
        })
    }

    // 将请求携带的信息和 OffsetAndMetadata 对象都交给
    // GroupCoordinator.handleCommitOffsets() 方法进行处理
    coordinator.handleCommitOffsets(offsetCommitRequest.groupId,
        offsetCommitRequest.memberId, offsetCommitRequest.generationId,
        partitionData, sendResponseCallback)
}
}
}

```

GroupCoordinator.handleCommitOffsets() 方法首先会进行一系列的检测, 保证 GroupCoordinator 处于可用状态且是对应 Consumer Group 的管理者, 与前面几个请求类似。这里多出了一种特殊情况的处理, 即 Consumer Group 的分区分配结果不由 GroupCoordinator 管理和保存, 例如在消费者客户端使用 KafkaConsumer.assign() 方法手动指定了消费者与分区之间的消费关系, 此时 GroupCoordinator 仅为其记录 offset 信息。然后, 根据 Group 的状态、memberId、generationId 的检测结果进行相应的处理。

```

def handleCommitOffsets(groupId: String, memberId: String, generationId:
Int,
                                offsetMetadata: immutable.Map[TopicPartition,
OffsetAndMetadata],
                                responseCallback: immutable.Map[TopicPartition,
Short] => Unit) {
    var delayedOffsetStore: Option[DelayedStore] = None
    ..... // 检测 GroupCoordinator 是否启动 (略)
    ..... // GroupCoordinator 是否管理此 Consumer Group (略)
    ..... // GroupCoordinator 是否已经加载此 Consumer Group 对应的 Offsets Topic
           // 分区 (略)

```

```

// 如果对应的 GroupMetadata 对象不存在且 generationId<0, 则表示 GroupCoordinator
// 不维护 Consumer Group 的分区分配结果, 只记录提交的 offset 信息
val group = groupManager.getGroup(groupId)
if (group == null) {
    if (generationId < 0)
        delayedOffsetStore = Some(groupManager.prepareStoreOffsets(groupId,
memberId,
        generationId, offsetMetadata, responseCallback))
    else
        responseCallback(offsetMetadata.mapValues(_ => Errors.ILLEGAL_
GENERATION.code))
} else {
    group.synchronized {
        ..... // 如果遇到 Group 处于 Dead 或 AwaitingSync、未知的 memberId、不合法
        // 的 generationId, 则调用回调函数并返回错误码 (略)
        val member = group.get(memberId)
        completeAndScheduleNextHeartbeatExpiration(group, member) // 更新心跳
        // 将记录 offset 的消息追加到对应的 Offsets Topic 分区中
        delayedOffsetStore = Some(groupManager.prepareStoreOffsets(groupId,
            memberId, generationId, offsetMetadata, responseCallback))
    }
}
delayedOffsetStore.foreach(groupManager.store)
}

```

GroupMetadataManager.prepareStoreOffsets() 方法和 store() 方法在上一节已经分析过了, 这里不再重复描述。

LeaveGroupRequest 分析

当消费者离开 Consumer Group, 例如调用 unsubscribe() 方法取消对 Topic 的订阅时, 会向 GroupCoordinator 发送 LeaveGroupRequest。API 层中由 KafkaApis.handleLeaveGroupRequest() 方法负责 LeaveGroupRequest 处理, 它会将请求委托给 GroupCoordinator.handleLeaveGroup() 方法。GroupCoordinator.handleSyncGroup() 方法会将对应 MemberMetadata 的 isLeaving 字段设置为 true 并尝试完成相应的 DelayedHeartbeat, 之后将对应的 MemberMetadata 对象从 GroupMetadata 中删除。最后 Consumer Group 的状态决定是否进行 Consumer Group 状态切换, 是否尝试完成对应的 DelayedJoin。GroupCoordinator.handleLeaveGroup() 方法的实现如下:

```

def handleLeaveGroup(groupId: String, consumerId: String,
                    responseCallback: Short => Unit) {
    ..... // 检测 GroupCoordinator 是否启动 (略)
    ..... // GroupCoordinator 是否管理此 Consumer Group (略)
    ..... // GroupCoordinator 是否已经加载此 Consumer Group 对应的 Offsets Topic
           // 分区 (略)
    group synchronized {
        ..... // 如果遇到 Group 处于 Dead、未知的 memberId, 则调用回调函数并返回错误码 (略)
        val member = group.get(consumerId)
        // 将对应 MemberMetadata 的 isLeaving 字段设置为 true, 尝试完成相应的
        // DelayedHeartbeat
        removeHeartbeatForLeavingMember(group, member)
        // 调用 GroupCoordinator.onMemberFailure() 方法移除对应的 MemberMetadata 对象
        // 并完成状态变化
        onMemberFailure(group, member)
        responseCallback(Errors.NONE.code) // 调用回调函数
    }
}

```

onGroupLoaded 和 onGroupUnloaded

GroupCoordinator.onGroupLoaded() 方法是在 GroupCoordinator.handleGroupImmigration() 方法中传入 GroupMetadataManager.loadGroupsForPartition() 方法的回调函数, 当出现 GroupMetadata 重复加载时, 会调用它更新心跳。

```

private def onGroupLoaded(group: GroupMetadata) {
    group synchronized {
        group.allMemberMetadata.foreach( // 更新所有 Member 的心跳操作
            completeAndScheduleNextHeartbeatExpiration(group, _)
        )
    }
}

```

GroupCoordinator.onGroupUnloaded() 方法是在 GroupCoordinator.handleGroupEmigration() 方法中传入 GroupMetadataManager.removeGroupsForPartition() 方法的回调函数, 它会在 GroupMetadata 被删除前, 将 Consumer Group 状态转换成 Dead, 并根据之前的 Consumer Group 状态进行相应的清理操作。


```

private def onGroupUnloaded(group: GroupMetadata) {
  group.synchronized {
    val previousState = group.currentState
    group.transitionTo(Dead) // 切换到 Dead 状态
    previousState match { // 根据 Consumer Group 之前的状态进行处理
      case Dead =>
      case PreparingRebalance =>
        // 调用全部 Member 的 awaitingJoinCallback 回调函数，返回指定错误码
        for (member <- group.allMemberMetadata) {
          if (member.awaitingJoinCallback != null) {
            member.awaitingJoinCallback(
              joinError(member.memberId, Errors.NOT_COORDINATOR_FOR_GROUP.code))
            // 清空 awaitingJoinCallback 回调函数
            member.awaitingJoinCallback = null
          }
        }
        // awaitingJoinCallback 的变化，可能导致 DelayedJoin 满足条件，故进行尝试
        joinPurgatory.checkAndComplete(GroupKey(group.groupId))

        // Stable、AwaitingSync 两种状态的处理与 PreparingRebalance 状态类似
      case Stable | AwaitingSync =>
        for (member <- group.allMemberMetadata) {
          if (member.awaitingSyncCallback != null) {
            member.awaitingSyncCallback(Array.empty[Byte],
              Errors.NOT_COORDINATOR_FOR_GROUP.code)
            member.awaitingSyncCallback = null
          }
          // 尝试执行 DelayedHeartbeat
          heartbeatPurgatory.checkAndComplete(MemberKey(member.groupId,
            member.memberId))
        }
    }
  }
}

```

本节介绍了 GroupMetadataManager 底层使用 Offsets Topic 以消息的形式记录 Consumer Group 的 GroupMetadata 信息和 offset 信息，并在内存中进行了相应的缓存。通过分析 GroupCoordinatorRequest 的处理流程，明确了 GroupCoordinator 与 Consumer Group

之间的对应关系。当 Offsets Topic 分区的 Leader 副本发生迁移时, GroupCoordinator 会通过迁入、迁出操作变化其管理的 Consumer Group 集合。介绍了每个 Consumer Group 在服务端对应状态的含义, 以及状态转换的时机和条件。详细分析了 JoinGroupRequest、SyncGroupRequest、OffsetCommitRequest、OffsetFetchRequest、ListGroupRequest、HeartbeatRequest、LeaveGroupRequest 的处理流程和具体实现。详细分析了 DelayedJoin、DelayedHeartbeat 实现和使用。希望读者通过本节的阅读, 能够对 GroupCoordinator 组件的原理和实现有清晰的认识。

4.8 身份认证与权限控制

为了增加 Kafka 集群的安全性, 从 Kafka 的 0.9 版本开始提供“身份认证 (Authentication)”和“权限控制 (Authorization)”的功能。这两个功能其实很常见, 但也很容易混淆, 我们做简单区分, “身份认证”指的是客户端 (生产者或消费者) 通过某些凭据, 例如用户名 / 密码或是 SSL 证书, 让服务端确认客户端的真实身份。“权限控制”指的是服务端根据客户端的身份, 决定其对某些资源是否有某些操作权限。“身份认证”在 Kafka 中的具体体现是服务器是否允许与当前请求的客户端建立连接, “权限控制”则体现在对消息的读写等方面的权限上。在 Kafka 0.10 版本中, 客户端与服务端支持使用 SSL、SASL/PLAIN 等方式进行连接, 默认情况下, 这些“身份认证”方式是不开启的。

SSL (Secure Sockets Layer) 是一种基于传输层 (比如 TCP/IP) 或应用层 (比如 HTTP) 的协议。SSL 协议依赖于数字证书, 数字证书中的核心组成部分是私钥和公钥两部分。SSL 分为“握手协议”和“传输协议”两部分, 其中“握手协议”是基于非对称加密的, 而“传输协议”是基于对称加密的。“握手协议”的主要目的是为了在客户端与服务端之间协商和交换对称密钥。之所以这么做是因为非对称密钥的算法比较复杂, 速度较慢, 不适合对大量数据进行加密, 而相较之下, 对称加密速度较快, 适合对大量数据加密。无论是对称加密还是非对称加密, 使用之后都会影响系统的性能, 我们需要在安全性和吞吐量之间进行适当权衡。

SASL (Simple Authentication and Security Layer) 是一种用来扩充 C/S 模式验证能力的认证机制, 它定义了客户端和服务端之间数据的交换方式, 但是并没指定数据的内容。其中, SASL/PLAIN 是最简单的、也是最危险的机制, 因为用户名和密码都是以明文的形式在网络中传输的, 别人可以轻松地从网络中截取这些信息。一般情况, 在安全的网络环境下考虑使用此机制, 当然也可以将用户名密码进行加密以提高安全性。本节也将以 SASL/PLAIN 为例展开描述和分析。

Kafka 将其“权限控制”功能设计成插件形式，使用非常灵活，同时也提供了一个基于 ZooKeeper 实现的 Authorizer，它将所有的 ACLs（Access Control Lists，访问控制列表）信息保存在 ZooKeeper 中，在 ACLs 中指定了用户对某些资源有某些权限。

4.8.1 配置 SASL/PLAIN 认证

在开始介绍 Kafka 的具体实现之前，首先完成基于 SASL/PLAIN 认证方式的环境搭建，让读者对 Kafka 的身份认证和权限控制模块有个大体上的了解。

Kafka 服务端的配置

首先，需要对“config/server.properties”配置文件进行修改，修改内容如下：

```
listeners=SASL_PLAINTEXT://localhost:9092
security.inter.broker.protocol=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=PLAIN
sasl.enabled.mechanisms=PLAIN
authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer
super.users=User:admin
```

创建 kafka_server_jaas.conf 配置文件并将该文件放置在 myConf 目录下，其内容如下：

```
KafkaServer {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="admin"
    password="admin"
    user_admin="admin"
    user_xiaoming="xiaoming";
};
```

这里我们配置了两个用户：admin 和 xiaoming，密码分别为 admin 和 xiaoming。

在启动 Kafka 服务端时，将 kafka_server_jaas.conf 配置文件以 VM 参数的形式传递进去，命令如下：

```
-Djava.security.auth.login.config="D:/myConf/kafka_server_jaas.conf"
```

也可以考虑在 kafka-run-class.sh 中直接添加 VM 参数。

Kafka 客户端的配置

首先创建 `kafka_client_jaas.conf` 配置文件:

```
KafkaClient { #LogContextName
    org.apache.kafka.common.security.plain.PlainLoginModule required
    username="xiaoming"
    password="xiaoming"
};
```

之后,在客户端的程序中配置环境变量,具体实现如下:

```
// 环境变量添加,需要输入配置文件的路径
System.setProperty("java.security.auth.login.config",
    "D:/myConf/kafka_client_jaas.conf");
Properties props = new Properties();
// Kafka 服务端的主机名和端口号
props.put("bootstrap.servers", "localhost:9092");
props.put("security.protocol", "SASL_PLAINTEXT");
props.put("sas.l.mechanism", "PLAIN");
```

此时执行客户端,得到如下异常:

```
org.apache.kafka.common.errors.TopicAuthorizationException:
    Not authorized to access topics: [test]
```

默认情况下,不指定具体的权限就表示禁止此权限。这里并未给 `xiaoming` 用户分配读写“test”这个 Topic 的权限,默认是禁止的。使用 `kafka-acl` 脚本进行权限分配之后,客户端即可正常运行,具体命令如下:

```
kafka-acls.bat --authorizer-properties zookeeper.connect=localhost:2181
--add --allow-principal User: xiaoming --operation Read --operation Write
--topic test
```

权限配置示例

在客户端配置的最后,使用 `kafka-acls` 脚本为 `xiaoming` 用户设置了对 Topic `test` 的读写权限,Kafka 中除了读写权限,还有删除权限、创建权限、修改权限、获取描述信息的权限等。`kafka-acls` 脚本有多个参数,可以与权限组合得到多种可能的配置。本节简单介绍几组比较常用的权限配置命令。

下面使用 `kafka-acls` 命令为 `xiaoming` 和 `wanglei` 授予从 IP 为 192.168.1.103 和 192.168.1.104 两台机器上读写 `test` 这个 Topic 的权限：

```
kafka-acls.bat --authorizer-properties zookeeper.connect=localhost:2181
--add --allow-principal User:xiaoming --allow-principal User:wanglei
--allow-host 192.168.1.103 --allow-host 192.168.1.104 --operation Read
--operation Write --topic test
```

有些场景下，允许的 ACLs 信息非常多，但禁止的 ACLs 信息很少，我们可以使用 “`--deny-principal`” 指定那些需要禁止的 ACLs 信息。如下所示，此命令表示允许 `liming` 之外的所有人从 192.168.1.117 这个 IP 读取 `test` 这个 Topic 的消息。

```
kafka-acls.bat --authorizer-properties zookeeper.connect=localhost:2181
--add --allow-principal User:* --allow-host * --deny-principal
User:liming
--deny-host 192.168.1.117 --operation Read --topic test
```

我们可以使用 “`--remove`” 参数删除之前指定的某些权限。如下所示，此命令表示删除 `xiaoming` 和 `wanglei` 从 192.168.1.103 读取 Topic `test` 的权限。

```
kafka-acls.bat --authorizer-properties zookeeper.connect=localhost:2181
--remove --allow-principal User:xiaoming --allow-principal
User:wanglei
--allow-host 192.168.1.103 --operation Read --topic test
```

还可以使用 “`--list`” 参数查询 ACLs 信息。如下所示，此命令表示查询 `test` 这个 Topic 上的全部 ACLs 信息。

```
kafka-acls.bat --authorizer-properties zookeeper.connect=localhost:2181
--list --topic test
```

为了用户方便快速地完成权限配置，`kafka-acls` 脚本提供了 “`--producer`” 和 “`--consumer`” 这两个特殊的参数。“`--producer`” 参数表示指定用户可以像生产者一样使用 Topic，其本质是 `WRITE`、`DESCRIBE`、`CREATE` 等权限的集合。“`--consumer`” 表示用户可以像消费者一样使用 Topic，其本质是 `READ`、`DESCRIBE` 等权限的集合。示例命令如下：


```
kafka-acls.bat --authorizer-properties zookeeper.connect=localhost:2181
--add
--allow-principal User:xiaoming --producer --topic test

kafka-acls.bat --authorizer-properties zookeeper.connect=localhost:2181
--add
--allow-principal User:wanglei --consumer --topic test
```

4.8.2 身份认证

客户端

在介绍身份认证的相关配置时,熟悉 JAAS (Java Authentication Authorization Service) 的读者看到 `kafka_server_jaas.conf` 配置文件,应该知道 Kafka 使用了 JAAS 的相关内容。JAAS 在应用层与底层安全机制之间加入了一层抽象,简化了 Java Security 包之上的开发工作,可以为开发人员屏蔽掉具体使用的安全机制,而且当安全机制改变后,应用层也不需要修改安全相关的代码。上层应用的代码主要是面向 `LoginContext` 进行编程,在 `LoginContext` 下层是可动态配置的 `LoginModule` 组件,在 `LoginModule` 组件中封装了使用正确的安全机制进行验证的相关代码,其架构如图 4-78 所示。

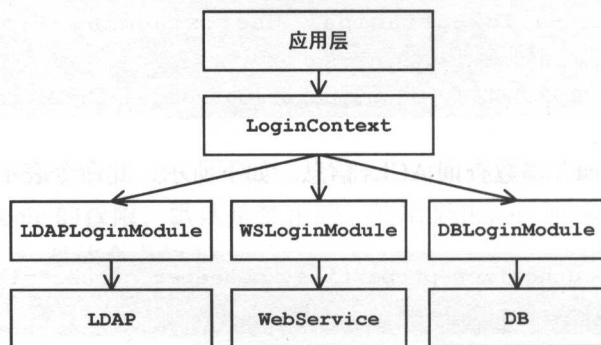


图 4-78

从前面介绍的 `kafka_client_jaas.conf` 配置文件中我们可以知道, Kafka 客户端使用的 `LoginModule` 是 Kafka 自定义的 `PlainLoginModule` 组件。

```

public class PlainLoginModule implements LoginModule {
    private static final String USERNAME_CONFIG = "username";
    private static final String PASSWORD_CONFIG = "password";

    // 下面这段静态代码块在客户端的作用不大，在分析服务端时再详细介绍
    static {
        PlainSaslServerProvider.initialize();
    }

    @Override
    public void initialize(Subject subject,
        CallbackHandler callbackHandler,
        Map<String, ?> sharedState, Map<String, ?> options) {
        String username = (String) options.get(USERNAME_CONFIG); // 读取 username
        if (username != null)
            subject.getPublicCredentials().add(username);
        String password = (String) options.get(PASSWORD_CONFIG); // 读取 password
        if (password != null)
            subject.getPrivateCredentials().add(password);
    }
    ..... // login() 方法、logout() 方法和 commit() 方法全部返回 true, abort() 方
        // 法返回 false (略)
}

```

网络上很多介绍 JAAS 的文章都提到在 LoginModule 中会完成具体的认证操作，但在 PlainLoginModule 组件中并没有具体的认证操作，仅仅是读取了配置文件中的用户名和密码并设置到 Subject 中。Subject 对象表示的是一个主体，也就是对资源访问的发起者，其中有三个字段 principals、pubCredentials、privCredentials，分别表示主体的身份、公钥/用户名、私钥/密码。

使用 JAAS 时，应用层的代码主要是操纵 LoginContext 对象，在 Kafka 中使用 Login 接口对 LoginContext 进行了又一次封装，其类间关系如图 4-79 所示。

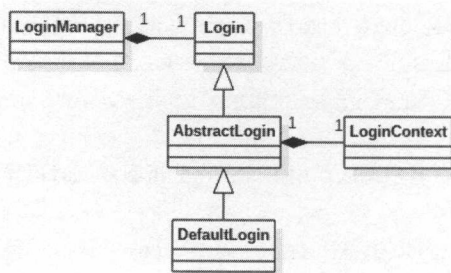


图 4-79

在 Login 接口中定义了如下四个方法：

```

public interface Login {
    // 配置 Login 对象
    void configure(Map<String, ?> configs, String loginContextName);

    // 调用 LoginModule 的 login() 方法和 commit() 方法
    LoginContext login() throws LoginException;

    // 返回 Subject 对象，即 PlainLoginModule 中配置好的 Subject 对象
    Subject subject();

    // 返回服务名称，在 DefaultLogin 实现中始终返回 “kafka” 字符串
    String serviceName();
}

```

在 SASL/PLAIN 身份认证的场景下，使用的是 DefaultLogin 实现，其具体实现都继承自 AbstractLogin 抽象类。AbstractLogin 中有 loginContextName 和 loginContext 两个字段，loginContextName 指明了使用 LoginContext 的名称，Kafka 客户端中该字段值为 KafkaClient，与 kafka_client_jaas.conf 配置文件第一行的 “KafkaClient” 匹配才能找到此 LoginContext。loginContext 字段是一个 LoginContext 对象，与 loginContextName 指定的名称对应，其中可以包含多个 LoginModule。当调用 LoginContext.login() 方法时，会依次调用所有的 LoginModule（这里的配置文件中只有 PlainLoginModule）对象的 login() 方法和 commit() 方法来完成认证操作。下面是 AbstractLogin 的具体实现代码：

```

public abstract class AbstractLogin implements Login {
    private String loginContextName;
    private LoginContext loginContext;

    public void configure(Map<String, ?> configs, String loginContextName) {
        this.loginContextName = loginContextName;
    }

    public LoginContext login() throws LoginException {
        String jaasConfigFile = System.getProperty(JaasUtils.JAVA_LOGIN_
CONFIG_PARAM);
        // 是否指定了 java.security.auth.login.config 配置
        if (jaasConfigFile == null) {
            log.debug("...");
        }
        AppConfigurationEntry[] configEntries = Configuration.
getConfiguration()
            .getAppConfigurationEntry(loginContextName);

        // 检测是否能找到“KafkaClient”这个 LoginContext 的配置
        if (configEntries == null) {
            String errorMessage = "...";
            throw new IllegalArgumentException(errorMessage);
        }
        // 创建 LoginContext 对象
        loginContext = new LoginContext(loginContextName, new
LoginCallbackHandler());
        loginContext.login(); // 调用 LoginContext.login() 方法，完成认证操作
        return loginContext;
    }

    // 返回 PlainLoginModule 中设置好用户名密码的 Subject 对象
    public Subject subject() { return loginContext.getSubject(); }
}

```

LogManager 中的核心逻辑就是创建根据配置创建 Login 的子类对象，并调用 login() 方法。代码比较简单，这里就不贴出来了。

分析到这里，我们了解到的代码只是从 JAAS 配置文件中读取了配置并填充 Subject 对象，没有进行实质性的认证操作。真正完成认证操作是通过 KafkaChannel 的 Authenticator 对象。

在前面的章节中，多次使用到 NetworkClient 这个组件来建立网络连接、发送请求、接收响应，NetworkClient 底层依赖 Selector 组件管理多个 KafkaChannel 对象，这些 KafkaChannel 都是通过 ChannelBuilder 创建出来的，其结构如图 4-80 所示。

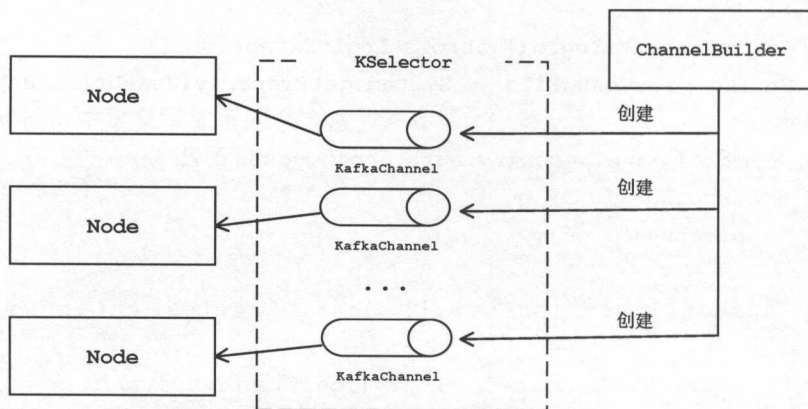


图 4-80

以 KafkaProducer 的构造方法为例，简单回顾一下 ChannelBuilder 和 KafkaChannel 的创建位置。

```

private KafkaProducer(ProducerConfig config, Serializer<K> keySerializer,
                      Serializer<V> valueSerializer) {
    ... ..
    // 根据配置的协议，创建不同的 ChannelBuilder 实现
    ChannelBuilder channelBuilder = ClientUtils
        .createChannelBuilder(config.values());
    NetworkClient client = new NetworkClient(
        new Selector(
            config.getLong(ProducerConfig.CONNECTIONS_MAX_IDLE_MS_
CONFIG),
            this.metrics, time, "producer", channelBuilder),
        this.metadata,
        clientId,
        config.getInt(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION),

```

```

config.getLong(ProducerConfig.RECONNECT_BACKOFF_MS_CONFIG),
config.getInt(ProducerConfig.SEND_BUFFER_CONFIG), config
.getInt(ProducerConfig.RECEIVE_BUFFER_CONFIG),
this.requestTimeoutMs, time);
.....
}

```

在 `Selector.connect()` 方法中会调用 `ChannelBuilder.buildChannel()` 方法创建 `KafkaChannel` 对象，之后会连接服务端进行交互。

```

public void connect(String id, InetSocketAddress address,
    int sendBufferSize, int receiveBufferSize) throws IOException {
    .....
    SelectionKey key = socketChannel.register(nioSelector, SelectionKey.OP_
CONNECT);
    KafkaChannel channel = channelBuilder.buildChannel(id, key,
maxReceiveSize);
    .....
}

```

回顾完之后继续对 `ChannelBuilder` 接口的分析，该接口有三个子类，分别实现了不同的身份认证方式，如图 4-81 所示。通过 `ClientUtils.createChannelBuilder()` 和 `ChannelBuilders.create()` 这两个方法，可以根据配置的协议选择使用对应的 `ChannelBuilder` 实现类。

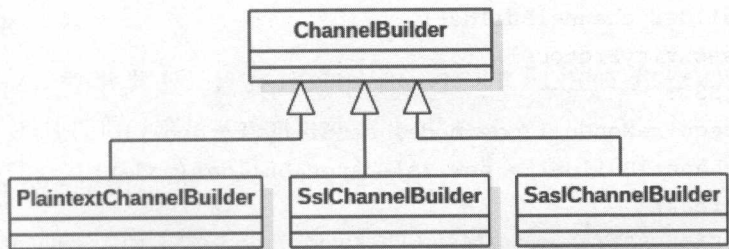


图 4-81

```

public static ChannelBuilder createChannelBuilder(Map<String, ?> configs) {
    // 根据“security.protocol”配置项的值，得到对应的 SecurityProtocol 对象
    SecurityProtocol securityProtocol = SecurityProtocol.forName(
        (String) configs.get(CommonClientConfigs.SECURITY_PROTOCOL_
CONFIG));

    if (!SecurityProtocol.nonTestingValues().contains(securityProtocol))
        throw new ConfigException("Invalid SecurityProtocol " +
securityProtocol);
    // 获取“saslmecanism”配置项的值
    String clientSaslMechanism = (String) configs.get(SaslConfigs.SASL_
MECHANISM);

    // 调用 ChannelBuilders.create() 方法，创建对应的 ChannelBuilder 对象
    return ChannelBuilders.create(securityProtocol, Mode.CLIENT, LoginType.
CLIENT,
        configs, clientSaslMechanism, true);
}

// 下面是 ChannelBuilders.create() 方法的代码
public static ChannelBuilder create(SecurityProtocol securityProtocol,
Mode mode,
    LoginType loginType, Map<String, ?> configs, String
clientSaslMechanism,
    boolean saslHandshakeRequestEnable) {
    ChannelBuilder channelBuilder;
    switch (securityProtocol) {
        case SSL:
            requireNonNullMode(mode, securityProtocol);
            channelBuilder = new SslChannelBuilder(mode);
            break;
        case SASL_SSL:
        case SASL_PLAINTEXT:
            // 在前面的示例中配置的是“SASL_PLAINTEXT”，所以创建 SaslChannelBuilder 对象
            requireNonNullMode(mode, securityProtocol);
            if (loginType == null)
                throw new IllegalArgumentException("...");
            if (mode == Mode.CLIENT && clientSaslMechanism == null)

```

```

        throw new IllegalArgumentException("...");
        channelBuilder = new SaslChannelBuilder(mode, loginType,
                                                securityProtocol, clientSaslMechanism,
saslHandshakeRequestEnable);
        break;
    case PLAINTEXT:
    case TRACE:
        channelBuilder = new PlaintextChannelBuilder();
        break;
    default:
        throw new IllegalArgumentException("...");
}
// 调用 channelBuilder.configure() 方法配置 ChannelBuilder 对象
channelBuilder.configure(configs);
return channelBuilder;
}

```

SaslChannelBuilder 中各个字段的含义如下所述。

- securityProtocol: 使用的安全协议, 也就是 security.protocol 配置项的值, 当前按照 SASL_PLAINTEXT 进行分析。
- clientSaslMechanism: 使用的 SASL 机制, 也就是 sasl.mechanism 配置项的值, 这里的值为 PLAIN。
- mode: 标识当前是客户端还是服务端, 对应的值分别是 Mode.CLIENT 和 Mode.SERVER。
- loginType: 枚举类型, 只有 LOGIN_CONTEXT_CLIENT 和 LOGIN_CONTEXT_Server, 具体的值分别是 KafkaClient 和 KafkaServer。
- handshakeRequestEnable: 是否发送握手消息。
- loginManager: 用于封装 LoginContext 的 LogManager 对象。
- configs: 配置信息。

在 ChannelBuilders.create() 方法中创建 SaslChannelBuilder 对象后, 紧接着就调用了 SaslChannelBuilder.config() 方法, 其中的核心操作就是创建 LoginManager 对象, 在 LoginManager 的构造方法中会创建 Login 对象并调用其 login() 方法。


```

public void configure(Map<String, ?> configs) throws KafkaException {
    try {
        this.configs = configs;
        boolean hasKerberos;
        ... ..
        if (hasKerberos) {
            ... .. // SASL/Kerberos 的相关处理 (略)
        }

        // 创建 LoginManager 对象, 其中会创建 DefaultLogin 对象并调用其 login() 方法,
        // 并最终调用
        // LoginContext.login() 方法
        this.loginManager = LoginManager.acquireLoginManager(loginType,
            hasKerberos, configs);
        if (this.securityProtocol == SecurityProtocol.SASL_SSL) {
            ... .. // SASL/SSL 的相关处理 (略)
        }
    } catch (Exception e) {
        throw new KafkaException(e);
    }
}

```

为了读者便于理解, ChannelBuilder 的创建以及初始化过程可以通过图 4-82 这张时序图进行总结。

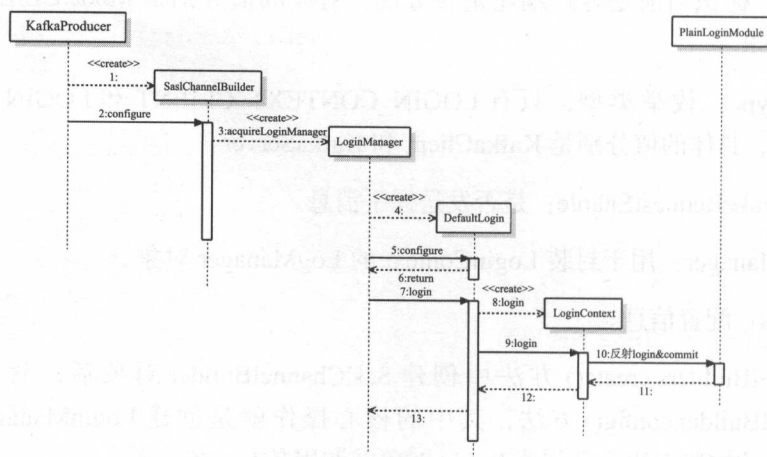


图 4-82

SaslChannelBuilder 主要负责创建 KafkaChannel 对象，该功能在 SaslChannelBuilder.buildChannel() 方法中实现。在 buildChannel() 方法中会创建 SaslClientAuthenticator 对象，并赋值给 KafkaChannel 中的 authenticator 字段，SaslClientAuthenticator 对象是完成身份认证的关键。

```
public KafkaChannel buildChannel(String id, SelectionKey key, int
maxReceiveSize)
    throws KafkaException {
    try {
        SocketChannel socketChannel = (SocketChannel) key.channel();
        // 创建 PlaintextTransportLayer 对象，它表示底层连接，其中封装
        // 了 SocketChannel 和 SelectionKey
        TransportLayer transportLayer = buildTransportLayer(id, key,
socketChannel);
        Authenticator authenticator;
        // 创建 SaslServerAuthenticator 对象，这是完成认证操作的关键
        if (mode == Mode.SERVER)
            authenticator = new SaslServerAuthenticator(id,
                loginManager.subject(), kerberosShortName,
                socketChannel.socket().getLocalAddress().
getHostName(),
                maxReceiveSize);
        else
            // 在客户端创建的是 SaslClientAuthenticator 对象
            authenticator = new SaslClientAuthenticator(id,
                loginManager.subject(), loginManager.serviceName(),
                socketChannel.socket().getInetAddress()
                    .getHostName(), clientSaslMechanism,
                handshakeRequestEnable);
        // 通过 configure() 方法将 TransportLayer 作为参数传递进去，在
        // SaslServerAuthenticator 中会与服务端进行通信，完成身份认证
        authenticator.configure(transportLayer, null, this.configs);
        return new KafkaChannel(id, transportLayer, authenticator,
maxReceiveSize);
    } catch (Exception e) {
        throw new KafkaException(e);
    }
}
```

Authenticator 接口中定义了 `authenticate()` 这个身份认证的关键方法，它有三个子类，如图 4-83 所示。其中 `DefaultAuthenticator.authenticate()` 方法为空实现，在前面的分析过程中没有进行身份认证时，使用的都是 `DefaultAuthenticator` 对象。`SaslServerAuthenticator` 和 `SaslClientAuthenticator` 分别是在服务端和客户端中使用的 `Authenticator` 对象。

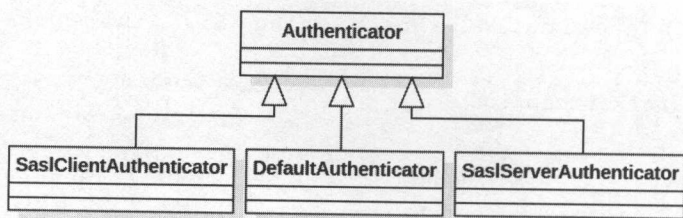


图 4-83

`SaslClientAuthenticator` 中各个字段的含义和功能如下，`SaslServerAuthenticator` 的相关内容在后面分析服务端身份认证时再详细介绍。

- **subject**: Subject 对象，表示用于身份认证的主体，通过 `SaslChannelBuilder.buildChannel()` 方法的分析，可知该对象就是前面 `PlainLoginModule` 初始化的 Subject 对象。
- **saslClient**: `javax.security` 包中提供的用于 SASL 身份认证的客户端接口。
- **transportLayer**: `PlaintextTransportLayer` 对象，该字段表示底层的网络连接，其中封装了 `SocketChannel` 和 `SelectionKey`。
- **netInBuffer**: `NetworkReceive` 对象，读取身份认证信息的输入缓冲区。
- **netOutBuffer**: `NetworkSend` 对象，发送身份认证信息的输出缓冲区。
- **saslState**: `SaslState` 类型，标识当前 `SaslClientAuthenticator` 的状态。
- **pendingSaslState**: `SaslState` 类型，在输出缓冲区中的内容全部清空前，由该字段暂存下一个 `saslState` 的值。
- **callbackHandler**: 用于收集身份认证信息的回调函数。

先来简单介绍一下 `SaslClient` 接口的使用。SASL 是一种 challenge-response 协议，读者可以将这种 challenge-response 协议理解成一种问答协议。服务端发布 Challenge（问题）到客户端，而客户端基于 Challenge 发送 Response（回答），当客户端返回的 Response 认证失败后，Server 可以继续发送 Challenge。其中 Challenge 和 Response 是任意长度的二进制标记。服务端与客户端交互的过程如图 4-84 所示。

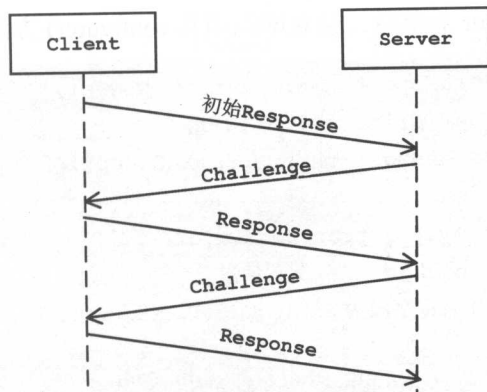


图 4-84

客户端的伪代码如下所示。

```

// callbackHandler 用于搜集认证信息
SaslClient sc = Sasl.createSaslClient(mechanisms, ..., callbackHandler);
String mechanism = sc.getName();
// 初始 Response
byte[] response = (sc.hasInitialResponse())?sc.evaluateChallenge(new
byte[]):null);
// 发送初始 Response
send(mechanism, response);
msg = receive(); // 获取服务端发来的 Challenge
while (!sc.isComplete()) { // 循环结束的条件是身份验证完成
    // 根据 Challenge 产生 Response
    response = sc.evaluateChallenge(msg.contents);
    if (msg.status == SUCCESS) { // 已经认证通过, 则结束循环
        if (response != null) { // 已经认证通过, 不应向服务端发送 Response
            throw new IOException("...");
        }
        break;
    } else { // 继续发送 Response, 并接收 Challenge
        send(mechanism, response);
        msg = receive();
    }
}
}

```

回到对 SaslClientAuthenticator 的分析, 在 SaslChannelBuilder.buildChannel() 方法中创

建完 SaslClientAuthenticator 对象后，会立即调用其 configure() 方法进行配置。

```
public void configure(TransportLayer transportLayer,
    PrincipalBuilder principalBuilder,
    Map<String, ?> configs) throws KafkaException {
    try {
        this.transportLayer = transportLayer; // PlaintextTransportLayer 对象
        this.configs = configs; // 配置信息
        // 初始化 saslState 字段为 SEND_HANDSHAKE_REQUEST, pendingSaslState 字
        // 段为 null
        setSaslState(handshakeRequestEnable ?
            SaslState.SEND_HANDSHAKE_REQUEST : SaslState.INITIAL);
        // determine client principal from subject.
        if (!subject.getPrincipals().isEmpty()) {
            Principal clientPrincipal = subject.getPrincipals().iterator().
next();
            this.clientPrincipalName = clientPrincipal.getName();
        } else {
            clientPrincipalName = null;
        }

        // 用于搜集认证信息的 SaslClientCallbackHandler
        callbackHandler = new SaslClientCallbackHandler();
        callbackHandler.configure(configs, Mode.CLIENT, subject, mechanism);

        saslClient = createSaslClient(); // 创建 SaslClient 对象
    } catch (Exception e) {
        throw new KafkaException("Failed to configure SaslClientAuthenticator", e);
    }
}
```

使用 SASL/PLAIN 进行身份认证时，创建的是 PlainClient 对象，它是 SaslClient 的实现类，SaslClient 接口有多个实现类，如图 4-85 所示。

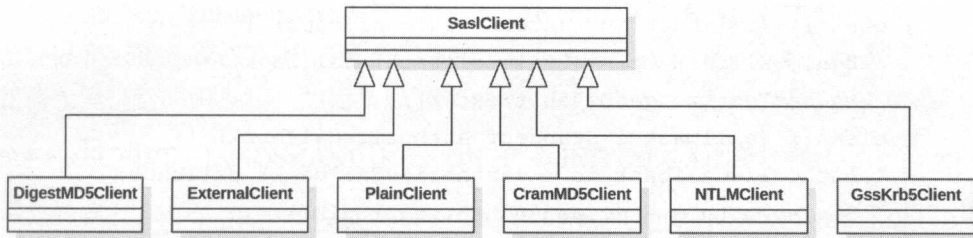


图 4-85

在创建过程中，会调用 `SaslClientCallbackHandler.handle()` 方法搜集认证需要的信息，并保存到 `PlainClient` 中。有的 `SaslClient` 实现是在 `evaluateChallenge()` 过程中进行认证信息搜集的，例如 `DigestMD5Client`。其他 `SaslClient` 的内容，这里不再赘述了，感兴趣的读者可以参考相关资料学习。

```

public void handle(Callback[] callbacks) throws UnsupportedCallbackException {
    for (Callback callback : callbacks) {
        if (callback instanceof NameCallback) {
            NameCallback nc = (NameCallback) callback;
            // 获取用户名
            if (!isKerberos && subject != null
                && !subject.getPublicCredentials(String.class).
isEmpty()) {
                nc.setName(subject.getPublicCredentials(String.class)
                    .iterator().next());
            } else
                nc.setName(nc.getDefaultName());
        } else if (callback instanceof PasswordCallback) {
            // 获取密码
            if (!isKerberos && subject != null
                && !subject.getPrivateCredentials(String.class).
isEmpty()) {
                char [] password = subject.getPrivateCredentials(String.
class)
                    .iterator().next().toCharArray();
                ((PasswordCallback) callback).setPassword(password);
            } else {
                throw new UnsupportedCallbackException(...);
            }
        }
    }
}
  
```

```

    } else if (callback instanceof RealmCallback) {
        RealmCallback rc = (RealmCallback) callback;
        rc.setText(rc.getDefaultText());
    } else if (callback instanceof AuthorizeCallback) {
        AuthorizeCallback ac = (AuthorizeCallback) callback;
        String authId = ac.getAuthenticationID();
        String authzId = ac.getAuthorizationID();
        ac.setAuthorized(authId.equals(authzId));
        if (ac.isAuthorized()) // 认证通过, 则设置 authorizedId
            ac.setAuthorizedID(authzId);
    } else {
        throw new UnsupportedCallbackException(callback, "...");
    }
}
}

```

最后, 我们来看一下 `SaslClientAuthenticator.authenticate()` 方法如何完成客户端身份认证的过程。`authenticate()` 方法的大致流程是: 首先, 发送握手消息, 确定服务端是否支持客户端使用的 SASL 机制 (SASL/PLAIN 对应的是 PLAIN); 然后, 发送一个空消息到服务端初始化身份认证流程; 之后, 通过 `SaslClient.evaluateChallenge()` 方法处理服务端的 Challenge 得到 Response, 并向服务端发送 Response, 重复此过程直到身份验证完成。

在图 4-86 中展示了 `authenticate()` 方法中 `saslState` 的状态转换。下面来分析 `SaslClientAuthenticator` 在进行状态转换时的行为。

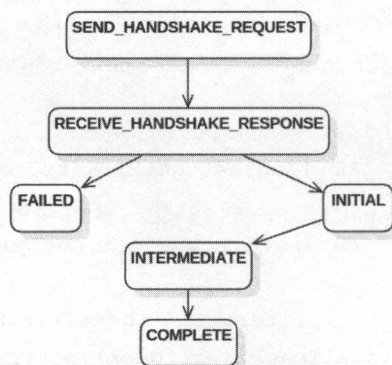


图 4-86

- SEND_HANDSHAKE_REQUEST → RECEIVE_HANDSHAKE_RESPONSE

当 SaslClientAuthenticator 处于 SEND_HANDSHAKE_REQUEST 状态时，会创建 SaslHandshakeRequest，其中记录了当前客户端使用的 SASL 机制（String 类型）。之后，将此请求发送给服务端，并切换成 RECEIVE_HANDSHAKE_RESPONSE 状态。

- RECEIVE_HANDSHAKE_RESPONSE → INITIAL/FAILED

此转换过程中，客户端会接收服务端返回的 SaslHandshakeResponse，在其中记录了 short 类型的错误码以及服务端支持的 SASL 机制集合。如果服务端返回了非零的错误码则抛出异常，并切换成 FAILED 状态；否则切换为 INITIAL 状态。

- INITIAL → INTERMEDIATE

此过程中会以一个空的 byte 数组作为初始 Response 发送给服务端，并切换为 INTERMEDIATE 状态。

- INTERMEDIATE → COMPLETE

此过程中会读取服务端返回的 Challenge，之后调用 saslClient.evaluateChallenge() 方法进行处理并得到 Response。如果已经验证通过则 Response 为空，不需要继续发送 Response；否则继续发送 Response。如果身份认证通过，则将状态切换为 COMPLETE，并且不再关注 TransportLayer 上的 OP_WRITE 事件；否则继续维持 INTERMEDIATE 状态。

SaslClientAuthenticator.authenticate() 方法的具体实现如下：

```
public void authenticate() throws IOException {
    // 发送缓冲区中还有未发送的数据，则需要先将这些数据发送完毕
    if (netOutBuffer != null && !flushNetOutBufferAndUpdateInterestOps())
        return;
    switch (saslState) {
        case SEND_HANDSHAKE_REQUEST:
            String clientId =
                (String) configs.get(CommonClientConfigs.CLIENT_ID_
CONFIG);
            currentRequestHeader = new RequestHeader(ApiKeys.SASL_
HANDSHAKE.id,
                clientId, correlationId++);
            // 创建并发送 SaslHandshakeRequest
            SaslHandshakeRequest handshakeRequest = new SaslHandshakeReque
st(mechanism);
            send(RequestSend.serialize(currentRequestHeader,
```



```

        handshakeRequest.toStruct()));
    // 切换成 RECEIVE_HANDSHAKE_RESPONSE 状态
    setSaslState(SaslState.RECEIVE_HANDSHAKE_RESPONSE);
    break;
case RECEIVE_HANDSHAKE_RESPONSE:
    // 读取 SaslHandshakeResponse
    byte[] responseBytes = receiveResponseOrToken();
    if (responseBytes == null) // 未读取到一个完整的消息
        break;
    else {
        try {
            // handleKafkaResponse() 方法主要负责解析 SaslHandshakeResponse,
            // 如果服务端返回了非零的错误码, 则抛出异常; 否则此方法正常返回
            handleKafkaResponse(currentRequestHeader, responseBytes);
            currentRequestHeader = null;
        } catch (Exception e) {
            setSaslState(SaslState.FAILED);
            throw e;
        }
        setSaslState(SaslState.INITIAL); // 切换成 INITIAL 状态
    }
    // 注意, 这里并没有 break, 还会继续执行下面 INITIAL 部分的代码
    // 如果下面的 sendSaslToken() 方法可以直接将数据发送完成,
    // 则可以减少一次调用 authenticate() 方法的次数
case INITIAL:
    // 发送空的 byte 数组, 初始化身份认证流程
    sendSaslToken(new byte[0], true);
    setSaslState(SaslState.INTERMEDIATE);
    break;
case INTERMEDIATE:
    // 读取服务端返回的 Challenge 信息
    byte[] serverToken = receiveResponseOrToken();
    if (serverToken != null) { // 读取到完整的 Challenge 信息
        sendSaslToken(serverToken, false); // 处理 Challenge 信息
    }
    // 如果身份认证通过, 则切换为 COMPLETE
    if (saslClient.isComplete()) {
        setSaslState(SaslState.COMPLETE);
    }

```

```

        transportLayer.removeInterestOps(SelectionKey.OP_WRITE);
    }
    break;
case COMPLETE: // 不做任何操作, 直接返回
    break;
case FAILED:
    throw new IOException("SASL handshake failed");
}
}

```

`SaslClientAuthenticator.authenticate()` 方法通过调用 `send()` 方法、`receiveResponseOrToken()` 方法、`sendSaslToken()` 方法三个方法完成了其功能。

`SaslClientAuthenticator.send()` 方法负责发送输出缓冲区中的数据, 检测缓冲区中的数据是否已全部发送完, 并根据检测结果设置在对应 `SocketChannel` 上关注的事件。在下次 `Selector.pollSelectionKeys()` 方法中会处理此 `SocketChannel` 上的被触发事件, `pollSelectionKeys()` 方法请读者参考第2章的分析。

```

private void send(ByteBuffer buffer) throws IOException {
    try {
        // 将待发送数据封装成 NetworkSend 对象
        netOutBuffer = new NetworkSend(node, buffer);
        flushNetOutBufferAndUpdateInterestOps();
    } catch (IOException e) {
        setSaslState(SaslState.FAILED);
        throw e;
    }
}

// 下面是 flushNetOutBufferAndUpdateInterestOps() 方法的实现
private boolean flushNetOutBufferAndUpdateInterestOps() throws IOException {
    boolean flushedCompletely = flushNetOutBuffer(); // 发送数据
    if (flushedCompletely) { // 如果全部发送完, 则取消对 OP_WRITE 事件的关注
        transportLayer.removeInterestOps(SelectionKey.OP_WRITE);
        if (pendingSaslState != null)
            setSaslState(pendingSaslState);
    } else // 如果未发送完, 则需要继续关注 OP_WRITE 事件
        transportLayer.addInterestOps(SelectionKey.OP_WRITE);
}

```

```

        return flushedCompletely;
    }

    // 下面是 flushNetOutBuffer() 方法的实现
    private boolean flushNetOutBuffer() throws IOException {
        if (!netOutBuffer.completed()) {
            netOutBuffer.writeTo(transportLayer); // 向 SocketChannel 中写数据
        }
        return netOutBuffer.completed();
    }
}

```

`SaslClientAuthenticator.receiveResponseOrToken()` 负责从 `SocketChannel` 中读取一个完成的消息。

```

private byte[] receiveResponseOrToken() throws IOException {
    // 创建缓冲区
    if (netInBuffer == null) netInBuffer = new NetworkReceive(node);
    netInBuffer.readFrom(transportLayer); // 从 SocketChannel 中读取数据
    byte[] serverPacket = null;
    // 如果读取到一个完成的消息，则返回读取的负载数据；否则返回 null
    if (netInBuffer.complete()) {
        netInBuffer.payload().rewind();
        serverPacket = new byte[netInBuffer.payload().remaining()];
        netInBuffer.payload().get(serverPacket, 0, serverPacket.length);
        netInBuffer = null; // 清空缓冲区
    }
    return serverPacket;
}

```

`SaslClientAuthenticator.sendSaslToken()` 方法负责处理服务端发送过来的 Challenge 信息，并将得到的新 Response 信息发送给服务端。

```

private void sendSaslToken(byte[] serverToken, boolean isInitial) throws
IOException {
    if (!saslClient.isComplete()) {
        // 处理 Challenge 信息
        byte[] saslToken = createSaslToken(serverToken, isInitial);
        if (saslToken != null)

```



```

        send(ByteBuffer.wrap(saslToken)); // send() 方法前面已经分析过了
    }
}

// 下面是 createSaslToken() 方法的实现
private byte[] createSaslToken(final byte[] saslToken, boolean isInitial)
    throws SaslException {
    try {
        // 初始 Response 的处理
        if (isInitial && !saslClient.hasInitialResponse())
            return saslToken;
        else
            return Subject.doAs(subject, new PrivilegedExceptionAction<byte[]>() {

                public byte[] run() throws SaslException {
                    // 调用 SaslClient.evaluateChallenge() 方法处理 Challenge 信息
                    return saslClient.evaluateChallenge(saslToken);
                }

            });
    } catch (PrivilegedActionException e) {
        ... .. // 重新抛出 SaslException 异常 (略)
    }
}

```

最后用一张时序图来总结 KafkaChannel 的创建过程以及验证过程，如图 4-87 所示。

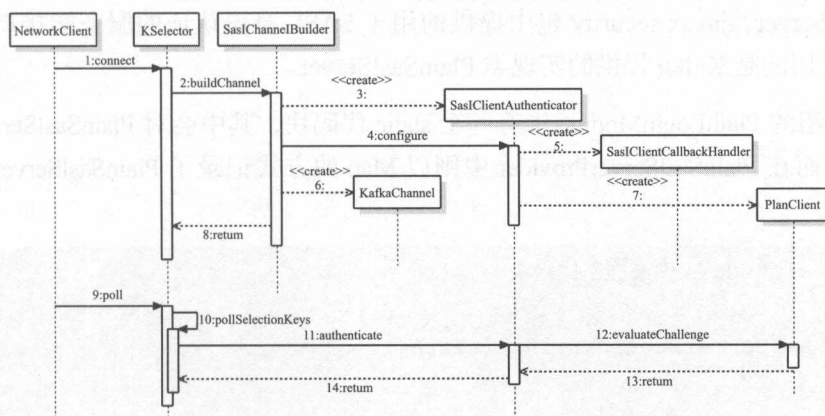


图 4-87

服务端

通过前面对 Kafka 网络层的介绍, 我们知道服务端也是使用 `NetworkClient` 组件来管理网络连接, `ChannelBuilder` 和 `KafkaChannel` 的创建过程与客户端相同。Kafka 在服务端实现 SASL/PLAIN 身份认证的方式也与客户端的类似, 也是在 `KafkaChannel.prepare()` 方法中通过调用 `Authenticator.authenticate()` 方法实现的, 但是使用的 `Authenticator` 接口的实现类不同。在 `SaslChannelBuilder.buildChannel()` 方法中创建 `KafkaChannel` 时, 有如下的代码片段:

```
if (mode == Mode.SERVER)
    authenticator = new SaslServerAuthenticator(id, loginManager.
subject(),
        kerberosShortNamer, socketChannel.socket().getLocalAddress().
getHostName(),
        maxReceiveSize);
else
    authenticator = new SaslClientAuthenticator(id, loginManager.
subject(),
        loginManager.serviceName(),
        socketChannel.socket().getInetAddress().getHostName(),
        clientSaslMechanism, handshakeRequestEnable);
```

服务端使用的 `Authenticator` 接口实现是 `SaslServerAuthenticator`, 其中的 `subject`、`transportLayer`、`netInBuffer`、`netOutBuffer`、`saslState` 等字段含义和功能与客户端使用的 `SaslClientAuthenticator` 相同, 这里不再赘述, 有两个与客户端不同的字段。

- `saslServer`: `javax.security` 包中提供的用于 SASL 身份认证的服务端接口, 在服务端使用的是 Kafka 提供的实现类 `PlainSaslServer`。

前面介绍的 `PlainLoginModule` 中有一个 `static` 代码块, 其中会对 `PlainSaslServerProvider` 进行注册, 而在 `PlainSaslServerProvider` 中则以 `Map` 的方式记录了 `PlainSaslServerFactory` 工厂类的名称。

```
// PlainLoginModule 中的静态代码块
static {
    PlainSaslServerProvider.initialize();
}

// PlainSaslServerProvider 的代码如下
```

```

public class PlainSaslServerProvider extends Provider {
    protected PlainSaslServerProvider() {
        // 记录 PlainSaslServerFactory 工厂类
        super.put("SaslServerFactory." + PlainSaslServer.PLAIN_MECHANISM,
            PlainSaslServerFactory.class.getName());
    }
    public static void initialize() {
        Security.addProvider(new PlainSaslServerProvider()); // 注册
    }
}

```

在后面的代码中如果需要使用 SaslServer，会通过 SaslServerAuthenticator.createSaslServer() 方法查找到 PlainSaslServerFactory 工厂类，并通过反射的方式创建其对象，然后创建 PlainSaslServer 对象。

```

private void createSaslServer(String mechanism) throws IOException {
    this.saslMechanism = mechanism;
    callbackHandler = new SaslServerCallbackHandler(
        Configuration.getConfiguration(), kerberosNamer);
    callbackHandler.configure(configs, Mode.SERVER, subject, saslMechanism);
    if (mechanism.equals(SaslConfigs.GSSAPI_MECHANISM)) {
        ... ..
    } else {
        try {
            saslServer = Subject.doAs(subject,
                new PrivilegedExceptionAction<SaslServer>() {
                    public SaslServer run() throws SaslException {
                        // 调用 Sasl.createSaslServer() 方法
                        return Sasl.createSaslServer(saslMechanism,
                            "kafka", host, configs,
                            callbackHandler);
                    }
                });
        } catch (PrivilegedActionException e) {
            throw new SaslException("...", e.getCause());
        }
    }
}

```

```
// 下面是 Sasl.createSaslServer() 方法的实现
public static SaslServer createSaslServer(String mechanism)
    throws SaslException {
    ... ..
    String mechFilter = "SaslServerFactory." + mechanism;
    // 查找 PlainSaslServerProvider
    Provider[] provs = Security.getProviders(mechFilter);
    for (int j = 0; provs != null && j < provs.length; j++) {
        className = provs[j].getProperty(mechFilter); // 查找工厂类
        ... ..
        // 使用反射方式创建 PlainSaslServerFactory 对象
        fac = (SaslServerFactory) loadFactory(provs[j], className);
        if (fac != null) {
            // 使用 PlainSaslServerFactory 工厂对象创建 PlainSaslServer 对象
            mech = fac.createSaslServer(mechanism, protocol,
                serverName, props, cbh);
            if (mech != null) {
                return mech;
            }
        }
    }
    return null;
}
```

- **enabledMechanisms**: Set 集合, 记录了服务端支持的 SASL 机制。

还有一个需要注意的字段是 **saslState**, 其初始值为 **GSSAPI_OR_HANDSHAKE_REQUEST**。

SaslServerAuthenticator.configure() 方法主要用来初始化 **enabledMechanisms** 字段的内容。

```
public void configure(TransportLayer transportLayer,
    PrincipalBuilder principalBuilder, Map<String, ?>
    configs) {
    this.transportLayer = transportLayer; // 初始化 transportLayer 字段
    this.configs = configs; // 初始化配置信息
    // 读取配置信息, 初始化 enabledMechanisms 字段
    List<String> enabledMechanisms =
        (List<String>) this.configs.get(SaslConfigs.SASL_ENABLED_
    MECHANISMS);
```



```

if (enabledMechanisms == null || enabledMechanisms.isEmpty())
    throw new IllegalArgumentException("No SASL mechanisms are
enabled");
this.enabledMechanisms = new HashSet<>(enabledMechanisms);
}

```

在 `SaslServerAuthenticator.authenticate()` 方法中完成了 `saslState` 的状态切换以及服务端的验证操作。图 4-88 展示了此过程中 `SaslServerAuthenticator` 的状态转换。

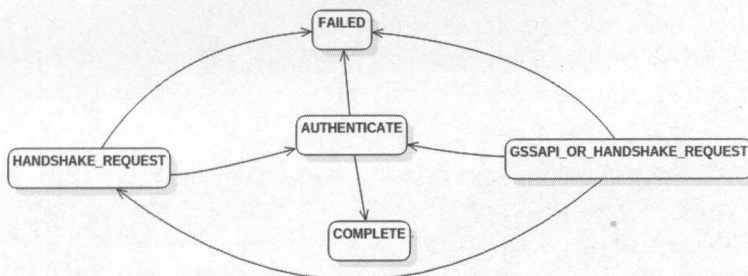


图 4-88

`authenticate()` 方法的大致步骤是：首先处理客户端发来的 `SaslHandshakeRequest`，验证服务端是否支持请求中指定的 SASL 机制，向客户端返回 `SaslHandshakeResponse`，并将状态切换为 `AUTHENTICATE`。之后，处理客户端发来的 `Response` 信息，如果身份认证失败，则生成相应的 `Challenge` 信息返回给客户端；如果成功，则不返回任何数据，只是将状态切换成 `COMPLETE`。上述过程中出现任何异常，都会将状态切换成 `FAILED` 并抛出异常。`SaslServerAuthenticator.authenticate()` 方法的具体实现如下：

```

public void authenticate() throws IOException {
    // 如果输出缓冲区中有未发送完成的数据，则先将这些数据发送出去
    if (netOutBuffer != null && !flushNetOutBufferAndUpdateInterestOps())
        return;
    if (saslServer != null && saslServer.isComplete()) { // 检测认证是否已完成
        setSaslState(SaslState.COMPLETE);
        return;
    }
    // 创建输入缓冲区，并从 SocketChannel 中读取数据
    if (netInBuffer == null) netInBuffer = new NetworkReceive(maxReceiveSize,
node);
    netInBuffer.readFrom(transportLayer);
}

```



```

if (netInBuffer.complete()) { // 检测是否读取了一个完整的消息
    netInBuffer.payload().rewind();
    byte[] clientToken = new byte[netInBuffer.payload().remaining()];
    // 获取消息负载
    netInBuffer.payload().get(clientToken, 0, clientToken.length);
    netInBuffer = null; // 重置输入缓冲区
    try {
        switch (saslState) {
            // HANDSHAKE_REQUEST 和 GSSAPI_OR_HANDSHAKE_REQUEST 状态都会处理握手消息
            case HANDSHAKE_REQUEST:
                handleKafkaRequest(clientToken);
                break;
            case GSSAPI_OR_HANDSHAKE_REQUEST:
                if (handleKafkaRequest(clientToken))
                    break;
            case AUTHENTICATE:
                // 调用 PlainSaslServer.evaluateResponse() 方法处理客户端发来的
                // Response 信息
                byte[] response = saslServer.evaluateResponse(clientToken);
                if (response != null) { // 返回 Challenge 信息
                    netOutBuffer = new NetworkSend(node, ByteBuffer.
wrap(response));

                    // 下面的 flushNetOutBufferAndUpdateInterestOps() 方法与
                    // SaslClientAuthenticator 中的同名方法实现类似, 不再赘述
                    flushNetOutBufferAndUpdateInterestOps();
                }
                if (saslServer.isComplete()) // 身份认证成功, 切换状态
                    setSaslState(SaslState.COMPLETE);
                break;
            default:
                break;
        }
    } catch (Exception e) {
        setSaslState(SaslState.FAILED); // 出现异常, 切换状态
        throw new IOException(e);
    }
}
}

```

在 `SaslServerAuthenticator.handleKafkaRequest()` 方法中除了检测 mechanism、切换状态，还会在握手成功后根据指定的 SASL 机制创建 `SaslServer`，即 `PlainSaslServer`。`handleKafkaRequest()` 方法的另一个功能是在身份认证未通过之前，完成 `ApiVersionsRequest` 的处理，当身份认证通过之后则由 `KafkaApis` 来完成。返回的 `ApiVersionResponse` 中记录了错误码、Broker 支持的所有 `APIKeys` 以及对应的版本号范围。

```
private boolean handleKafkaRequest(byte[] requestBytes)
    throws IOException, AuthenticationException {
    String clientMechanism = null;
    try {
        ByteBuffer requestBuffer = ByteBuffer.wrap(requestBytes);
        RequestHeader requestHeader = RequestHeader.parse(requestBuffer);
        ApiKeys apiKey = ApiKeys.forId(requestHeader.apiKey());
        setSaslState(SaslState.HANDSHAKE_REQUEST); // 状态切换
        ... .. // 检测 apiKey 和 version 是否合法
        switch (apiKey) {
            case API_VERSIONS: // 返回 Broker 支持的所有 APIKeys
                handleApiVersionsRequest(requestHeader, (ApiVersionsRequest)
request);
                break;
            // 检测服务端是否支持客户端指定的 mechanism，并返回响应
            case SASL_HANDSHAKE:
                clientMechanism = handleHandshakeRequest(requestHeader,
                    (SaslHandshakeRequest) request);
                break;
            default: ... .. // 抛出异常（略）
        }
    }
} catch (SchemaException | IllegalArgumentException e) {
    ... .. // 异常处理（略）
}
if (clientMechanism != null) {
    createSaslServer(clientMechanism); // 如果 SASL 机制检测通过，则创建
PlainSaslServer
    setSaslState(SaslState.AUTHENTICATE); // 切换成 AUTHENTICATE 状态
}
return isKafkaRequest;
}
```

在 `handleApiVersionsRequest()` 方法和 `handleHandshakeRequest()` 方法中都会调用 `sendKafkaResponse()` 方法将响应发送给客户端，具体实现与 `SaslClientAuthenticator.send()` 方法类似，这里就不再将代码贴出了。

`PlainSaslServer.evaluateResponse()` 方法实现了对客户端的 `Response` 信息的处理，具体实现如下：

```
public byte[] evaluateResponse(byte[] response) throws SaslException {
    String[] tokens;
    try {
        tokens = new String(response, "UTF-8").split("\u0000"); // 解析
    } catch (UnsupportedEncodingException e) {
        throw new SaslException("UTF-8 encoding not supported", e);
    }
    if (tokens.length != 3) throw new SaslException("..." + tokens.
length);
    authorizationID = tokens[0];
    String username = tokens[1];
    String password = tokens[2];
    ..... // 检测 username 和 password, 若为空, 则抛出异常 (略)

    // 在使用 SASL/PLAIN 方式进行身份认证时, authorizationID 为空, 会被赋值为
    // username, 在后面权限控制时, 会通过该字段确定其权限
    if (authorizationID.isEmpty()) authorizationID = username;
    try {
        // 读取配置文件中的信息, JAAS_USER_PREFIX 字段的值为 "user_"
        String expectedPassword = JaasUtils.jaasConfig(LoginType.SERVER.
contextName(),
            JAAS_USER_PREFIX + username);
        if (!password.equals(expectedPassword)) { // 检测密码是否正确
            throw new SaslException("...");
        }
    } catch (IOException e) {
        throw new SaslException("...");
    }
    complete = true;
    return new byte[0];
}
```


4.8.3 权限控制

通过前面对 Kafka 中多种请求的格式分析我们知道，客户端发送的请求中并没有封装任何身份信息，服务端是根据什么查找客户端的权限呢？通过上一小节对身份认证实现的分析可知，在每个 KafkaChannel 对象中都封装了一个 SocketChannel 对象和一个 PlainSaslServer 对象，身份认证成功后，在 PlainSaslServer 对象的 authorizationID 字段中记录了认证成功用户名。SocketChannel 与 authorizationID 通过这种方式一一对应，服务端就可以知道请求来自哪个 KafkaChannel，从而可以确定发送请求的客户端的身份。

在 Kafka 网络层的 Processor.processCompletedReceives() 方法中，会通过上述方式确定客户端的身份，并且将客户端的身份封装成 Session 对象，与读取到的请求信息一起放入 RequestChannel 中等待 Handler 线程处理。该代码片段如下所示。

```
private def processCompletedReceives() {
  selector.completedReceives.asScala.foreach { receive =>
    try {
      // 查找请求来自哪个 KafkaChannel
      val channel = selector.channel(receive.source)
      // 从 KafkaChannel 中封装的 SaslServerAuthenticator 对象中
      // 获取 authorizationID 信息，并封装成 Session
      val session = RequestChannel.Session(new KafkaPrincipal(
        KafkaPrincipal.USER_TYPE, channel.principal.getName), channel.
        socketAddress)
      // 将 Session 与请求数据封装成 RequestChannel.Request 对象
      val req = RequestChannel.Request(processor = id, connectionId =
        receive.source,
        session = session, buffer = receive.payload, startTimeMs = time.
        milliseconds,
        securityProtocol = protocol)
      // 将 Request 对象放入 RequestChannel，等待处理
      requestChannel.sendRequest(req)
    } catch {
      ... .. // 异常处理（略）
    }
  }
}
```

明确了客户端身份后，由 Authorizer 接口以插件的形式完成权限控制的功能。

Kafka 在启动时会读取配置文件中的“authorizer.class”配置项，并使用反射创建指定的 Authorizer 对象，然后调用其 configure() 方法配置此对象；如果“authorizer.class”配置项未指定，则不进行权限控制。创建并配置 Authorizer 对象的代码片段在 KafkaServer.startup() 方法中：

```
authorizer = Option(config.authorizerClassName)
    .filter(_ != null).map { authorizerClassName =>
    // 通过反射方式初始化“authorizer.class”配置项指定的 Authorizer 对象
    val authZ = CoreUtils.createObject[Authorizer](authorizerClassName)
    // 调用 Authorizer.configure() 方法进行配置
    authZ.configure(config.originals())
    authZ
  }
```

之后 Authorizer 对象会被传递给 KafkaApis 对象，Handler 线程处理各个请求是通过调用 KafkaApis.handle*() 方法实现的，其中都会调用 KafkaApis.authorize() 方法进行权限控制，调用栈如图 4-89 所示。KafkaApis.authorize() 方法会调用 Authorizer.authorize() 方法。

```
private def authorize(session: Session, operation: Operation,
    resource: Resource): Boolean =
    authorizer.map(_._authorize(session, operation, resource)).
    getOrElse(true)
```

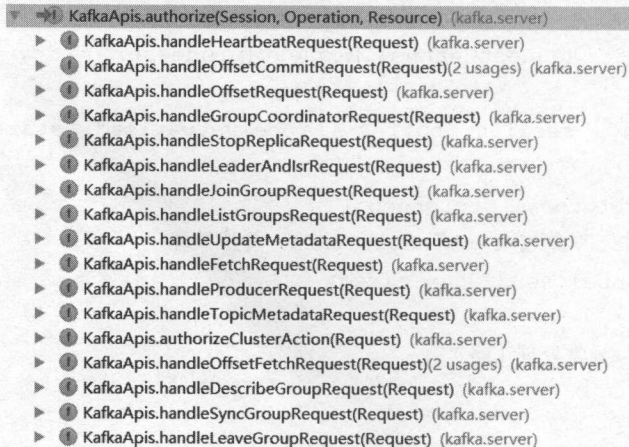


图 4-89

SimpleAclAuthorizer 由 Kafka 提供的 Authorizer 接口实现，它将权限信息储存在

ZooKeeper 中。SimpleAclAuthorizer 中各个字段的含义和功能如下所述。

- `superUsers`: `Set[KafkaPrincipal]` 集合, 用于记录配置文件中指定的超级用户的信息, 其中每一项都是 `KafkaPrincipal` 类型, `KafkaPrincipal` 中使用 `principalType` 字段记录客户端的身份类型, 本节示例中该值全部为 “User”; 使用 `name` 字段记录了客户端的身份, 即 `username`。
- `shouldAllowEveryoneIfNoAclIsFound`: 默认值 `false` 时表示, 如果一个资源在 ACLs 中没有相关记录时, 除了超级用户, 任何用户都不能访问。可以通过 “`allow.everyone.if.no.acl.found`” 配置项修改默认行为。
- `aclChangeListener`: ZooKeeper 上的监听器, 后面详述。
- `aclCache`: `HashMap[Resource, VersionedAcls]` 类型, ACLs 在内存中的缓存。

在前面的配置过程中, 使用 `kafka-acls` 脚本为 `xiaoming` 分配了 `test` 这个 Topic 的读写权限。此命令执行完成后, 可以在 ZooKeeper 的客户端中查看 “`/kafka-acl/Topic/test`” 节点的内容, 发现相应的权限设置如下所示。将权限的描述进行组装, 可以得到其含义 “Principal P is [Allowed/Denied] Operation O From Host H On Resource R”。

```
{
  "version": 1,
  "acls": [
    {
      "principal": "User:xiaoming",
      "permissionType": "Allow",
      "operation": "Read",
      "host": "*"
    },
    {
      "principal": "User:xiaoming",
      "permissionType": "Allow",
      "operation": "Write",
      "host": "*"
    }
  ]
}
```

在 `SimpleAclAuthorizer.configure()` 方法中会初始化很多信息, 例如: 超级用户的信息、`shouldAllowEveryoneIfNoAclIsFound` 字段、初始化 `ZkNodeChangeNotificationListener` 监听

器加载 aclCache 信息。该方法实现如下：

```

override def configure(javaConfigs: util.Map[String, _]) {
  val configs = javaConfigs.asScala
  val props = new java.util.Properties() // 对配置信息进行转换
  configs.foreach { case (key, value) => props.put(key, value.toString) }
  // 获取超级用户的信息
  superUsers = configs.get(SimpleAclAuthorizer.SuperUsersProp).collect {
    case str: String if str.nonEmpty => str.split(";").map(
      s => KafkaPrincipal.fromString(s.trim)).toSet
  }.getOrElse(Set.empty[KafkaPrincipal])

  // 根据配置, 初始化 shouldAllowEveryoneIfNoAclIsFound 字段
  shouldAllowEveryoneIfNoAclIsFound = configs.get(SimpleAclAuthorizer
    .AllowEveryoneIfNoAclIsFoundProp).exists(_._toString.toBoolean)

  val kafkaConfig = KafkaConfig.fromProps(props, doLog = false)
  // 获取 ZooKeeper 的地址、连接超时时间、Session 的过期时间等配置
  val zkUrl = configs.get(SimpleAclAuthorizer.ZkUrlProp).map(_._toString)
    .getOrElse(kafkaConfig.zkConnect)
  val zkConnectionTimeoutMs = configs.get(SimpleAclAuthorizer.
    ZkConnectionTimeoutProp)
    .map(_._toString.toInt).getOrElse(kafkaConfig.zkConnectionTimeoutMs)
  val zkSessionTimeoutMs = configs.get(SimpleAclAuthorizer.
    ZkSessionTimeoutProp)
    .map(_._toString.toInt).getOrElse(kafkaConfig.zkSessionTimeoutMs)
  // 创建 ZKUtil, 用于与 ZooKeeper 进行交互
  zkUtils = ZkUtils(zkUrl, zkConnectionTimeoutMs, zkSessionTimeoutMs,
    JaasUtils.isZkSecurityEnabled())

  // 检测 “/kafka-acl” 这个持久节点在 ZooKeeper 中是否存在, 若不存在则创建
  zkUtils.makeSurePersistentPathExists(SimpleAclAuthorizer.AclZkPath)

  loadCache() // 将 ZooKeeper 中的 ACLs 信息加载到 aclCache 集合中
  // 检测 “/kafka-acl-changes” 节点在 ZooKeeper 中是否存在, 若不存在则创建
  zkUtils.makeSurePersistentPathExists(SimpleAclAuthorizer.AclChangedZkPath)

  // ZooKeeper 的监听器, 后面详述

```



```

aclChangeListener = new ZkNodeChangeNotificationListener(zkUtils,
    SimpleAclAuthorizer.AclChangedZkPath, SimpleAclAuthorizer.
AclChangedPrefix,
    AclChangedNotificationHandler)
aclChangeListener.init()
}

```

SimpleAclAuthorizer.loadCache() 方法负责遍历 ZooKeeper 的 “/kafka-acl” 节点中记录的 ACLs 信息，并将其加载到 aclCache 集合。

```

private def loadCache() {
    inWriteLock(lock) { // 加锁保护 aclCache 集合的相关操作
        // 获取 “/kafka-acl” 的子节点集合，此集合表示的资源类型，例如：Topic、Cluster、
        // ConsumerGroup
        val resourceTypes = zkUtils.getChildren(SimpleAclAuthorizer.AclZkPath)
        for (rType <- resourceTypes) {
            // 转换成 ResourceType 类型
            val resourceType = ResourceType.fromString(rType)
            val resourceTypePath = SimpleAclAuthorizer.AclZkPath + "/" +
resourceType.name
            // 获取 “/kafka-acl/[ResourceType]” 下的子节点集合，此集合表示具体的资源，例如：
            // 示例中的 test 就属于 Topic 类型，所以在 “/kafka-acl/Topic” 节点下
            val resourceNames = zkUtils.getChildren(resourceTypePath)
            for (resourceName <- resourceNames) {
                // 遍历每个资源，对应节点中的 JSON 数据，并转换成 VersionedAcls
                val versionedAcls =
                    getAclsFromZk(Resource(resourceType, resourceName.toString))
                // 将 VersionedAcls 对象保存到 aclCache 集合中
                updateCache(new Resource(resourceType, resourceName),
versionedAcls)
            }
        }
    }
}

```

VersionedAcls 类中封装了 Acl 对象集合和 zkVersion 信息，Acl 类中的字段与 ZooKeeper 中记录的 ACLs 信息一一对应，其定义如下：


```
private case class VersionedAcls(acls: Set[Acl], zkVersion: Int)

case class Acl(principal: KafkaPrincipal, permissionType: PermissionType,
               host: String, operation: Operation){...}
```

其中, `PermissionType` 有 `Allow` 和 `Deny` 两种取值, `Operation` 有 `Read`、`Write`、`Create`、`Delete`、`Alter`、`Describe`、`ClusterAction`、`All` 这八种取值。`Resource` 类中封装了资源的类型 (`ResourceType`) 和资源的名称 (`String` 类型), `ResourceType` 有 `Cluster`、`Topic`、`Group` 三个子类, 表示三种不同类型的资源, 如图 4-90 所示。

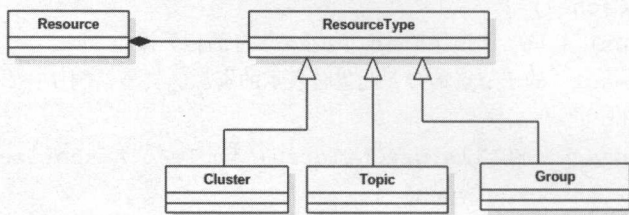


图 4-90

当管理人员使用 `kafka-acls` 脚本增删 ACLs 信息时, 实际上是调用 `AclCommand` 中的对应方法, 最终会将调用 `SimpleAclAuthorizer` 的对应方法完成对 `ZooKeeper` 的写入操作, 这些方法主要完成对 `ZooKeeper` 的读写等操作, 感兴趣的读者可以参考源码。

这里来关注 `ZkNodeChangeNotificationListener`, 它负责将 `ZooKeeper` 中 ACLs 信息的修改同步到 `aclCache` 集合中, 大致原理是: `ZkNodeChangeNotificationListener` 会在 `ZooKeeper` 的 `/kafka-acl-changes` 节点上注册 `NodeChangeListener`, 用来监听其子节点的变化。当通过 `kafka-acks` 命令增删 ACLs 信息时, 除了修改 `/kafka-acl` 路径下的 ACLs 数据, 还会在 `/kafka-acl-changes` 路径下添加一个持久顺序节点, 节点名称的前缀是 `“acl_changes_”` 字符串, 该节点中记录的数据是修改的资源类型和资源名称。之后 `NodeChangeListener` 会被触发, 它会根据节点名称重新加载相应资源的 ACLs 信息到 `aclCache` 集合中。

`ZkNodeChangeNotificationListener` 中各字段的含义如下所述。

- `seqNodeRoot`: 指定监听的路径, 这里的值是 `“/kafka-acl-changes”`。
- `seqNodePrefix`: 持久顺序节点的前缀, 这里的值为 `“acl_changes_”`。
- `notificationHandler`: 当监听到 `seqNodeRoot` 路径下子节点集合发生变化时, 执行的

响应操作。

- `changeExpirationMs`: 如果顺序节点创建后, 超过 `changeExpirationMs` 指定的时间, 则认为可以被删除, 默认值为 15 分钟。
- `lastExecutedChange`: 记录上次处理的顺序节点的编号。

在 `ZkNodeChangeNotificationListener.init()` 方法中会完成注册监听器的操作, 实现如下:

```
def init() {
    // 确保 "/kafka-acl-changes" 节点存在
    zkUtils.makeSurePersistentPathExists(seqNodeRoot)
    // 注册 NodeChangeListener, 监听 "/kafka-acl-changes" 子节点的变化
    zkUtils.zkClient.subscribeChildChanges(seqNodeRoot, NodeChangeListener)
    // 注册 ZkStateChangeListener, 监听与 ZooKeeper 的连接状态变化
    zkUtils.zkClient.subscribeStateChanges(ZkStateChangeListener)
    // 处理 "/kafka-acl-changes" 的子节点
    processAllNotifications()
}
```

当 `NodeChangeListener` 和 `ZkStateChangeListener` 被触发时, 都会调用 `processAllNotifications()` 方法处理 `"/kafka-acl-changes"` 的子节点。

```
def processAllNotifications() {
    val changes = zkUtils.zkClient.getChildren(seqNodeRoot)
    processNotifications(changes.asScala.sorted)
}

// processNotifications() 方法实现如下
private def processNotifications(notifications: Seq[String]) {
    if (notifications.nonEmpty) {
        try {
            val now = time.milliseconds
            for (notification <- notifications) { // 遍历子节点集合
                val changeId = changeNumber(notification) // 获取子节点编号
                if (changeId > lastExecutedChange) { // 检测此子节点是否已经处理过
                    val changeZnode = seqNodeRoot + "/" + notification
                    // 读取节点状态信息和其中记录的数据
                    val (data, stat) = zkUtils.readDataMaybeNull(changeZnode)
                    // 调用 NotificationHandler.processNotification() 方法更新 aclCache 集合
                }
            }
        } catch {
            case e: Exception => {
                // 异常处理
            }
        }
    }
}
```

```

        data map (notificationHandler.processNotification(_)) getOrElse
            (logger.warn("..."))
    }
    lastExecutedChange = changeId // 记录处理的子节点编号
}
purgeObsoleteNotifications(now, notifications) // 删除过期节点
} catch {
    ..... // 异常处理 (略)
}
}
}

// 下面是 purgeObsoleteNotifications() 方法的实现
private def purgeObsoleteNotifications(now: Long, notifications: Seq[String])
{
    for (notification <- notifications.sorted) {
        val notificationNode = seqNodeRoot + "/" + notification
        // 读取节点状态信息和其中记录的数据
        val (data, stat) = zkUtils.readDataMaybeNull(notificationNode)
        if (data.isDefined) {
            if (now - stat.getCtime > changeExpirationMs) { // 检测节点是否过期
                zkUtils.deletePath(notificationNode) // 删除节点
            }
        }
    }
}
}

```

在 `NotificationHandler.processNotification()` 方法中完成了更新 `aclCache` 集合的操作，`NotificationHandler` 对象在 `ZkNodeChangeNotificationListener` 创建时传入，这里使用的是 `AclChangedNotificationHandler` 实现。

```

object AclChangedNotificationHandler extends NotificationHandler {
    override def processNotification(notificationMessage: String) {
        val resource: Resource = Resource.fromString(notificationMessage)
        inWriteLock(lock) {
            // 从 ZooKeeper 读取指定的资源的 ACLs 信息
            val versionedAcls = getAclsFromZk(resource)

```



```

        updateCache(resource, versionedAcls) // 将新的 VersionedAcls 更新到
aclCache 集合中
    }
}
}

```

在 SimpleAclAuthorizer.authorize() 方法中会将传入的客户端对应的身份信息以及请求操作的资源信息与上述 aclCache 集合匹配, 决定是否有权限操作相应资源。具体实现如下:

```

override def authorize(session: Session, operation: Operation,
                        resource: Resource): Boolean = {
    val principal = session.principal // 获取用户身份, 这里是 username
    val host = session.clientAddress.getHostAddress
    // 获取指定资源的 ACLs 信息
    val acls = getAcls(resource) ++ getAcls(
        new Resource(resource.resourceType, Resource.WildCardResource))
    // 检测是否存在 Deny 类型的 ACLs 信息
    val denyMatch = aclMatch(session, operation, resource, principal, host,
        Deny, acls)

    // 如果有 Read 和 Write 权限, 则默认提供 Describe 权限
    val ops = if (Describe == operation)
        Set[Operation](operation, Read, Write)
    else
        Set[Operation](operation)

    // 检测是否存在 Allow 类型的 ACLs 信息
    val allowMatch = ops.exists(operation =>
        aclMatch(session, operation, resource, principal, host, Allow, acls))
    // 检测是否是超级管理员, 检测是否开启了 shouldAllowEveryoneIfNoAclIsFound,
    // 检测之前的匹配是否成功
    val authorized = isSuperUser(operation, resource, principal, host) ||
        isEmptyAclAndAuthorized(operation, resource, principal, host, acls) ||
        (!denyMatch && allowMatch)

    authorized
}

```



```
// 下面是 aclMatch() 方法的实现
private def aclMatch(session: Session, operations: Operation, resource:
Resource,
    principal: KafkaPrincipal, host: String, permissionType:
PermissionType,
    acls: Set[Acl]): Boolean = {
acls.find(acl =>
    acl.permissionType == permissionType // 匹配 PermissionType
    && (acl.principal == principal ||
        acl.principal == Acl.WildCardPrincipal) // 匹配身份信息以及通配符
    // 匹配操作以及通配符
    && (operations == acl.operation || acl.operation == All)
    // 匹配主机名以及通配符
    && (acl.host == host || acl.host == Acl.WildCardHost)
).map { acl: Acl =>
    true
}.getOrElse(false)
}
```

本节介绍了 SASL/PLAIN 和 JAAS 的基础知识和基本使用，介绍了 SaslClient 接口和 SaslServer 接口的使用，以及 Kafka 中如何实现 SASL/PLAIN 方式的身份认证。介绍了 Kafka 中的 Authorizer 接口以及 SimpleAclAuthorizer 实现，分析了基于 ZooKeeper 的权限控制的原理和实现。希望读者通过本章阅读，了解 Kafka 身份认证和权限控制的原理。

4.9 Kafka 监控

在大型的分布式系统中，Kafka 可能只是其中的一个组件，监控其运行状态是一项非常重要的工作。通过监控 Kafka 的各方面指标，管理人员可以知道 Kafka 是否成为了整个系统的瓶颈，也可以根据监控收集到的信息进行一系列优化操作，例如：优化 Kafka 配置参数、优化 JVM 参数以及升级操作系统和硬件等。根据的 Kafka 运行状况，也可以侧面反映出其 Kafka 上下游系统的运行状态。

在实践中，Kafka 会产生多种级别的日志，当系统出现故障的时候，可以根据日志对故障进行处理。同样也可以对日志进行统计和分析后作为调优依据，这也是常见的优化手段。但这种依赖日志的排故和优化方式毕竟是事后（发生故障或成为瓶颈之后）的分析，而且日志是静态的，无法反应系统的当前运行状态。有的场景中可能需要了解 Kafka 的实

时运行情况,例如,当前系统的连接数多少,系统处理了多少请求,这些请求中有多少得到了正常的响应,等等。在生产环境中,一般会将这两种手段配合使用,这样可以更快速地发现定位故障和瓶颈。

Kafka 中使用 Yammer Metrics 工具包进行内部状态的监控,默认通过 JMX 方式对外提供监控数据。在开始介绍 Kafka 中监控相关代码实现之前,先来介绍一下 JMX 和 Metrics 工具包的相关概念和使用方式。

4.9.1 JMX 简介

当系统功能比较简单、接口比较少的时候,开发人员可以通过与监控系统之间制定协议来进行通信。随着系统的功能不断丰富、规模不断扩大,会进行业务拆分等架构上的改变,出现多个子系统和独立的服务。如果依然使用上述方式开发监控模块、暴露监控数据,监控系统就需要了解每个服务的细节。为了解决这个问题,JMX 提供了一层抽象,屏蔽了获取监控数据的实现细节,如图 4-91 所示,监控系统只需要与 JMX 这一层抽象交互即可。

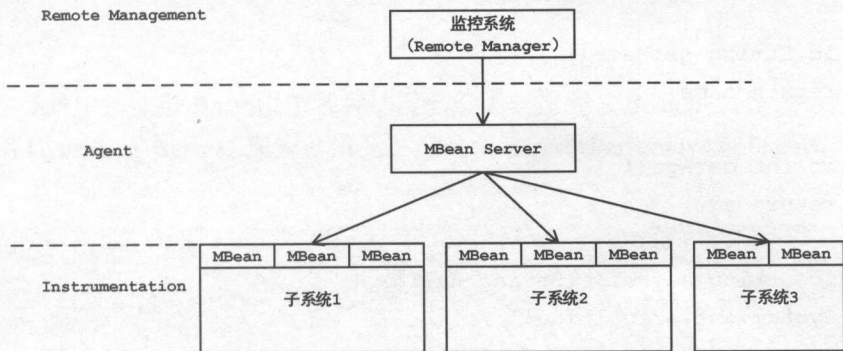


图 4-91

JMX 规范可以分为三层: Instrumentation、Agent、Remote Management。Instrumentation 层主要包括了一系列的接口定义和描述如何开发 MBean 的规范,这一层通过 MBean 封装 JMX 所管理的资源。Agent 层用于管理资源并为提供访问的接口,Agent 层的关键组件是 MBeanServer,其中还可以提供多个 Connector 和 Adapter 供外部访问。RemoteManagement 层是通过 Agent 访问资源的远端。

下面通过几个示例帮助读者快速入门,先来介绍 Standard MBean 的使用。每个 MBean 必须实现一个接口且接口的名称一般以“MBean”结尾,本例对应 PersonMBean 接口。该接口中定义了两个可读的属性 name 和 age 以及一个可执行的操作 sayHello()。实现如下:

```
public interface PersonMBean {
    public String getName();
    public int getAge();
    public String sayHello(String hello);
}
```

下面定义一个 **Person** 类实现 **PersonMBean** 接口，具体代码如下：

```
public class Person implements PersonMBean {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String sayHello(String hello) {
        System.out.println(hello);
        return this.name + ":" + hello;
    }
}
```

之后写一个 **PersonAgent** 类，将 **Person** 对象注册到 **MBeanServer** 中。

```
public class PersonAgent {
    public static void main(String[] args) throws Exception {
        // 获取 MBeanServer 对象
        MBeanServer server = ManagementFactory.getPlatformMBeanServer();
        ObjectName personName = new ObjectName("jmxBean:name=xiaoming");
        server.registerMBean(new Person("xiaoming", 27), personName); // 注册 MBean
        Thread.sleep(60 * 60 * 1000);
    }
}
```


启动 PersonAgent 之后, 可以通过 JConsole 连接到此程序, 在其 MBean 标签页中找到上面定义的名为“xiaoming”的 MBean, 其中有 Name 和 Age 两个可读字段和一个 sayHello() 操作, 可以为 sayHello() 操作指定参数并调用, 如图 4-92 所示。

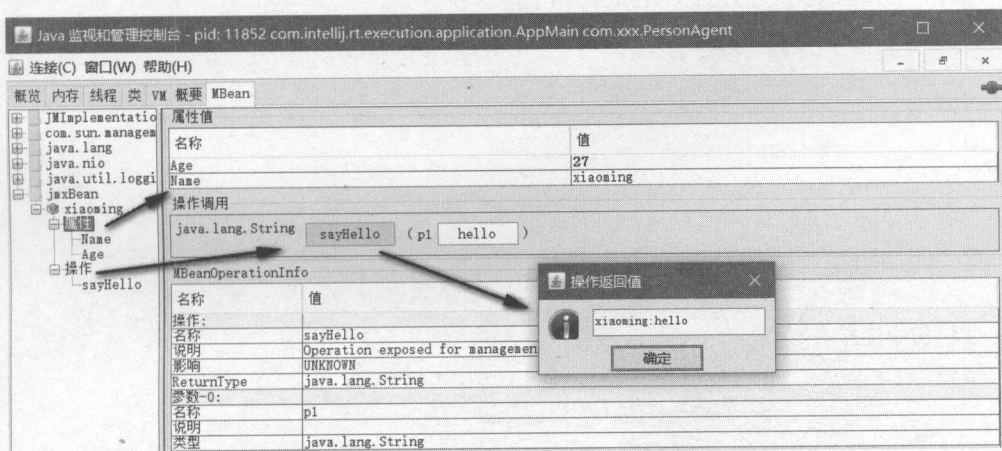


图 4-92

在有些场景中, 业务 Bean 由于种种情况已经无法按照 MBean 的规范进行修改了, 我们可以使用 DynamicMBean 处理这种情况。这里创建一个 PersonDynamic 类实现 Dynamic 接口。

```
public class PersonDynamic implements DynamicMBean {
    // Person 对象
    private Person person;

    // 描述属性信息
    private List<MBeanAttributeInfo> attributes = new ArrayList<MBeanAttributeInfo>();

    // 描述构造器信息
    private List<MBeanConstructorInfo> constructors =
        new ArrayList<MBeanConstructorInfo>();

    // 描述方法信息
    private List<MBeanOperationInfo> operations = new ArrayList<MBeanOperationInfo>();
    // 描述通知信息
    private List<MBeanNotificationInfo> notifications =
        new ArrayList<MBeanNotificationInfo>();
```



```

// MBeanInfo 用于管理以上描述信息
private MBeanInfo mBeanInfo;

public PersonDynamic(Person person) {
    this.person = person;
    try {
        init();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// 初始化方法
private void init() throws Exception {
    // 构建 Person 的属性、方法、构造器等信息
    constructors.add(new MBeanConstructorInfo("PersonDynamic(String, Integer)" +
        " 构造器", this.person.getClass().getConstructors()[0]));
    attributes.add(new MBeanAttributeInfo("name", "java.lang.String",
" 姓名 ",
        true, false, false));
    attributes.add(new MBeanAttributeInfo("age", "int", " 年龄 ",
        true, false, false));
    operations.add(new MBeanOperationInfo("sayHello() 方法 .", this.
person
        .getClass().getMethod("sayHello", new Class[]{String.
class})));

    // 创建一个 MBeanInfo 对象
    this.mBeanInfo = new MBeanInfo(this.getClass().getName(),
        "PersonDynamic",
        attributes.toArray(new MBeanAttributeInfo[attributes.
size()]),
        constructors.toArray(new MBeanConstructorInfo[constructors.size()]),
        operations.toArray(new MBeanOperationInfo[operations.
size()]),
        notifications.toArray(new MBeanNotificationInfo[notifications.size()])
    );
}

```

```

@Override
public Object getAttribute(String attribute) throws AttributeNotFoundException,
    MBeanException, ReflectionException { // 获取 person 对象属性值
    if (attribute.equals("name")) {
        return this.person.getName();
    } else if (attribute.equals("age")) {
        return this.person.getAge();
    }
    return null;
}

@Override
public AttributeList getAttributes(String[] attributes) {
    // 通过属性名获取一个属性对象列表
    if (attributes == null || attributes.length == 0) {
        return null;
    }
    try {
        AttributeList attrList = new AttributeList();
        for (String attrName : attributes) {
            Object obj = this.getAttribute(attrName);
            Attribute attribute = new Attribute(attrName, obj);
            attrList.add(attribute);
        }
        return attrList;
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

@Override
public MBeanInfo getMBeanInfo() { // 获取 MBeanInfo
    return mBeanInfo;
}

@Override
public Object invoke(String actionName, Object[] params, String[]
signature)

```

```

        throws MBeanException, ReflectionException {
    // 调用 Person 里面指定的方法
    if (actionName.equals("sayHello")) {
        return this.person.sayHello(params[0].toString());
    }
    return null;
}
// setAttribute() 方法和 setAttributes() 也是 DynamicMBean 接口需要实现的方法,
// 本例为空实现, 代码不再贴出来了
}

```

注册 DynamicMBean 的方式与 StandardMBean 一样, 也可以使用 JConsole 查看 MBean, 这里就不再赘述了。JMX 的基础知识就介绍到这里。

4.9.2 Metrics 简介

Metrics 是一个度量工具包, 提供多种度量类型来统计程序的各项指标。Metrics 的使用也比较简单, 只要按照其 API 将使用的度量类型嵌入到代码中即可。Metrics 默认支持并开启了 JMX 的方式暴露监控数据, 开发人员可以使用 JMX 的方式轻松获取度量数据。同时, Metrics 中还提供了其他类型 Reporter, 例如 CsvReporter, 它会周期性地 will 度量数据追加到指定的 CCSV 文件中, 感兴趣的读者可以参考 Metrics 的相关配置文档。本小节来介绍 Metrics 提供的五种度量类型及其使用方式。

Gauge 是 Metrics 中最简单的度量类型, 它用于记录瞬时值, 只有一个简单的返回值, 示例如下所示。

```

public static void main(String[] args) throws Exception {
    final List<String> list = new ArrayList<String>();
    ConsoleReporter.enable(5, TimeUnit.SECONDS); // 每隔 5 秒钟在控制台输出一次

    // 定义一个名为 "testGauge" 的 Gauge, 用于度量 list 集合的长度
    Gauge<Integer> g = Metrics.newGauge(Main3.class, "testGauge", new
    Gauge<Integer>() {
        public Integer value () {
            return list.size();
        }
    });
    while (true) {

```



```

        list.add("s");
        Thread.sleep(1000);
    }
}

```

----- 输出 -----

```

17-1-15 19:39:59 =====
=====
com.xxx.Main3: testGauge: value = 7
17-1-15 19:40:04 =====
=====
com.xxx.Main3: testGauge: value = 12
.....

```

Metrics 默认开启了 JmxReporter，在上例运行过程中，也可以通过 JConsole 查看到对应的 MBean 信息。

Counter 与 Gauge 类似，也用来度量数字值，但是可以通过 inc()、dec()、clear() 等方法修改其度量值，一般用来记录某事件的发生次数或是请求的个数。Counter 底层是通过 AtomicLong 实现的，示例代码如下：

```

public class MainCounter {
    // 创建 Counter 对象
    private final Counter testCounter = Metrics.newCounter(MainCounter.
class,
        "testCounter");
    private final List<String> list = new ArrayList<String>();

    public void add(String str) {
        testCounter.inc(); // 增加 Counter 度量的值
        list.add(str);
    }

    public String take() {
        testCounter.dec(); // 减小 Counter 度量的值
        return list.remove(0);
    }
}

```



```

public static void main(String[] args) throws Exception {
    MainCounter tc = new MainCounter();
    ConsoleReporter.enable(1, TimeUnit.SECONDS); // 每一秒输出一次
    while (true) {
        tc.add("s");
        Thread.sleep(1000);
    }
}

```

----- 输出 -----

```

17-1-15 21:48:09 =====
=====
com.xxx.MainCounter:
  testCounter:
    count = 22

```

Meter 是用来度量某时间段内平均请求数的，通过调用 Meter.mark() 方法表示收到一个请求。Meter 的最终统计结果有：请求数总数、平均每秒的请求数，以及最近的 1、5、15 分钟的平均 TPS。

```

public class MainMeter {
    private static Meter meter =
Metrics.newMeter(MainMeter.class, "Meter", "requests", TimeUnit.SECONDS);

    public static void main(String[] args) throws InterruptedException {
        ConsoleReporter.enable(1, TimeUnit.SECONDS);
        while (true) {
            meter.mark(); // 调用一次 mark() 方法就认为收到一次请求数
            meter.mark();
            Thread.sleep(1000);
        }
    }
}

```

----- 输出 -----

```

17-1-15 21:44:09 =====
=====

```

```
com.xxx.MainMeter:
Meter:count = 76; mean rate = 2.02 Meter/s; 1-minute rate = 2.00 Meter/s;
5-minute rate = 2.00 Meter/s; 15-minute rate = 2.00 Meter/s
... ..
```

Histogram 可以用于度量最大值、最小值、平均值、方差、中位值、百分比数据（例如 75%、90%、98%、99% 的数据在哪个范围内），示例如下：

```
public class MainHistogram {
    // 创建 Histogram 对象
    private static Histogram histo =
        Metrics.newHistogram(MainHistogram.class, "testHistogram");

    public static void main(String[] args) throws InterruptedException {
        ConsoleReporter.enable(1, TimeUnit.SECONDS);
        int i = 0;
        while (true) {
            histo.update(i++); // 更新 Histogram 度量数据
            Thread.sleep(1000);
        }
    }
}
```

----- 输出 -----

```
17-1-15 21:50:49 =====
=====
com.xxx.MainHistogram: testHistogram:
min = 0.00 max = 8.00 mean = 4.00 stddev = 2.74 median = 4.00 75% <= 6.50
95% <= 8.00 98% <= 8.00 99% <= 8.00 99.9% <= 8.00
... ..
```

Timer 主要用来统计某一块代码段的执行时间以及其分布情况，底层是基于 Histogram 和 Meter 来实现的，示例代码如下：



```

public class MainTimer {
    private static Timer timer = Metrics.newTimer(MainTimer.class, "testTimer",
        TimeUnit.MILLISECONDS, TimeUnit.SECONDS); // 创建 Timer 对象

    public static void main(String[] args) throws InterruptedException {
        ConsoleReporter.enable(1, TimeUnit.SECONDS);
        Random rn = new Random();
        timer.time();
        System.out.println();
        while (true) {
            TimerContext context = timer.time(); // 开始度量
            Thread.sleep(rn.nextInt(1000));
            context.stop(); // 结束度量
        }
    }
}

```

----- 输出 -----

```

17-1-15 21:57:21 =====
=====
com.xxx.MainTimer: testTimer:
count = 18   mean rate = 1.62 calls/s   1-minute rate = 1.42 calls/s   5-minute
rate = 1.40 calls/s   15-minute rate = 1.40 calls/s   min = 77.38ms   max
= 981.38ms   mean = 617.26ms   stddev = 246.20ms   median = 601.73ms   75% <=
840.44ms   95% <= 981.38ms   98% <= 981.38ms   99% <= 981.38ms   99.9% <= 981.38ms
... ..

```

介绍完 Metrics 中常用的度量类型以及其使用方式后，简单来分析一下 Metrics 对 JMX 支持的实现，读者了解即可。图 4-93 展示与 JmxReporter 相关的类之间的关系。

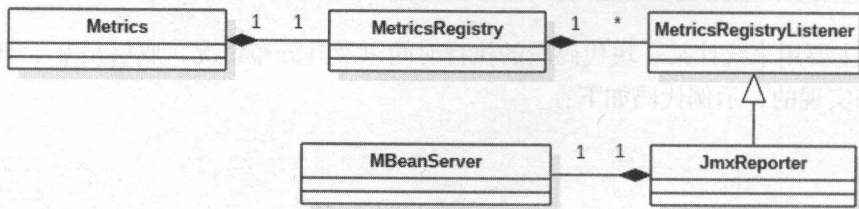


图 4-93

这里以 Gauge 的使用为例进行分析。当加载 Metrics 类时会执行其静态代码块，创建 MetricsRegistry 对象、JmxReporter 对象、MBeanServer 对象，其中 JmxReporter 对象以监听器的形式注册到 MetricsRegistry 上。该过程的相关实现如下：

```
public class Metrics { // Metrics 类
    // 创建 MetricsRegistry 对象
    private static final MetricsRegistry DEFAULT_REGISTRY = new
MetricsRegistry();
    static {
        JmxReporter.startDefault(DEFAULT_REGISTRY);
    }
    ... ..
}

// 下面是 JmxReporter 的代码，JmxReporter 继承了 MetricsRegistryListener，其中还
// 定义了前面介绍的五种度量类对应的 MBean 接口（略）
public class JmxReporter extends AbstractReporter implements
MetricsRegistryListener,
    MetricProcessor<JmxReporter.Context> {
    private final MBeanServer server;
    private static JmxReporter INSTANCE; // 单例模式
    // 管理 MBean 对象的集合
    private final Map<MetricName, ObjectName> registeredBeans;
    ... ..
    public static void startDefault(MetricsRegistry registry) {
        INSTANCE = new JmxReporter(registry);
        INSTANCE.start();
    }
    public final void start() {
        // 以监听器的形式注册到 MetricsRegistry 上
        getMetricsRegistry().addListener(this);
    }

    public JmxReporter(MetricsRegistry registry) {
        ... ..
        // 初始化 MBeanServer
        this.server = ManagementFactory.getPlatformMBeanServer();
    }
}
```


当调用 `Metrics.newGauge()` 方法时会触发 `JmxReporter.processGauge()`，最终调用 `MBeanServer.registerMBean()` 完成 MBean 的注册。

```
public class JmxReporter extends AbstractReporter implements MetricsRegistryListener,
    MetricProcessor<JmxReporter.Context> {
    .....
    public void processGauge(MetricName name, Gauge<?> gauge, Context
context)
        throws Exception {
        registerBean(context.getMetricName(),
            new Gauge(gauge, context.getObjectNames(), context.
getObjectNames());
    }

    private void registerBean(MetricName name, MetricMBean bean, ObjectNames
objectNames)
        throws MBeanRegistrationException, OperationsException {
        if ( server.isRegistered(objectNames) ){
            server.unregisterMBean(objectNames);
        }
        server.registerMBean(bean, objectNames);
        registeredBeans.put(name, objectNames);
    }
}
```

`Metrics` 中还提供了 `CVSReporter` 以及前面见到的 `ConsoleReporter`，其实现原理与 `JmxReporter` 基本类似，这里就不再赘述了。在下一小节还会分析 `Kafka` 中提供的 `JmxReporter`，请读者注意区分对比。

4.9.3 Kafka 中的 Metrics

`Kafka` 在 `Metrics` 的基础上又进行了一次封装。在 `KafkaMetricsGroup` 中定义了一些基础方法。正如前面分析时看到的，其他类继承 `KafkaMetricsGroup` 即可使用其中的方法。下面是 `KafkaMetricsGroup` 注册 `Metrics` 中五种度量对象的方法。

```

trait KafkaMetricsGroup extends Logging {

  def newGauge[T](name: String, metric: Gauge[T],
    tags: scala.collection.Map[String, String] = Map.empty) =
    Metrics.defaultRegistry().newGauge(metricName(name, tags), metric)

  def newMeter(name: String, eventType: String, timeUnit: TimeUnit,
    tags: scala.collection.Map[String, String] = Map.empty) =
    Metrics.defaultRegistry().newMeter(metricName(name, tags), eventType,
    timeUnit)

  def newHistogram(name: String, biased: Boolean = true,
    tags: scala.collection.Map[String, String] = Map.empty) =
    Metrics.defaultRegistry().newHistogram(metricName(name, tags), biased)

  def newTimer(name: String, durationUnit: TimeUnit, rateUnit: TimeUnit,
    tags: scala.collection.Map[String, String] = Map.empty) =
    Metrics.defaultRegistry().newTimer(metricName(name, tags),
    durationUnit, rateUnit)

  def removeMetric(name: String,
    tags: scala.collection.Map[String, String] = Map.empty) =
    Metrics.defaultRegistry().removeMetric(metricName(name, tags))
}

```

`KafkaMetricsGroup.metricName()` 方法按照一定的规则生成度量对象的 `MetricName`，尤其要注意其中 `MBean` 的名称。

```

private def metricName(name: String,
  tags: scala.collection.Map[String, String] = Map.empty) = {
  val klass = this.getClass // 当前类的 Class 对象
  val pkg = if (klass.getPackage == null) "" else klass.getPackage.getName
  // 包名
  val simpleName = klass.getSimpleName.replaceAll("\\\\$", "") // 简单类名
  // 创建 MetricsName
  explicitMetricName(pkg, simpleName, name, tags)
}

```

```
// 下面是 explicitMetricName() 方法的实现
private def explicitMetricName(group: String, typeName: String, name:
String, tags: scala.collection.Map[String, String] = Map.empty) = {
    val nameBuilder: StringBuilder = new StringBuilder // 用于创建 MBean 的名称
    nameBuilder.append(group) // 第一部分是 group, 即包名
    nameBuilder.append(":type=") // 第二部是 type, 即类名
    nameBuilder.append(typeName)
    if (name.length > 0) { // 第三部分是 name
        nameBuilder.append(",name=")
        nameBuilder.append(name)
    }

    val scope: String = KafkaMetricsGroup.toScope(tags).getOrElse(null)
    // 遍历 tags 集合, 以逗号分隔每个 Entry, 形成字符串
    val tagsName = KafkaMetricsGroup.toMBeanName(tags)
    tagsName match {
        case Some(tn) =>
            nameBuilder.append(",").append(tn) // 第四部分是 tags
        case None =>
    }
    new MetricName(group, typeName, name, scope, nameBuilder.toString())
}
```

使用 `KafkaMetricsGroup` 的地方比较多, 下面对每个度量类型选择一处进行分析。在之前分析的 `Log` 类中, 可以看到 `Gauge` 这个度量类的使用方法, `Log` 类中相关的代码片段如下:

```
val tags = Map("topic" -> topicAndPartition.topic,
    "partition" -> topicAndPartition.partition.toString)

// 在 Log 类初始化时, 会执行下面的代码创建并注册 Gauge 对象
newGauge("NumLogSegments", new Gauge[Int] {
    def value = numberOfSegments // 记录当前 Log 中的 Segment 对象个数
}, tags)

def numberOfSegments: Int = segments.size
```

通过对 `KafkaMetricsGroup.metricName()` 方法的描述, 可以得到上述代码片段产生的

MBean 名称为: kafka.log:type=Log,name=NumLogSegments,topic=[Topic],partition=[partition]。通过 JConsole 可以找到该 MBean 的信息,如图 4-94 所示。

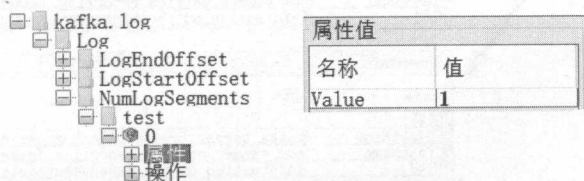


图 4-94

在 ReplicaManager 类中使用 Meter 度量类统计全部分区的 ISR 集合发生扩张 / 缩小的频率,相关代码片段如下:

```
val isrExpandRate = newMeter("IsrExpandsPerSec", "expands", TimeUnit.SECONDS)
val isrShrinkRate = newMeter("IsrShrinksPerSec", "shrinks", TimeUnit.SECONDS)
```

在 ReplicaManager.maybeExpandIsr() 方法中会调用 isrExpandRate 的 mark() 方法,标识发生了一次 ISR 集合扩张。

```
def maybeExpandIsr(replicaId: Int) {
    .....
    replicaManager.isrExpandRate.mark() // 标识发生了一次 ISR 集合扩张
    .....
}
```

在 ReplicaManager.maybeShrinkIsr() 方法中会调用 isrShrinkRate 对应的 mark() 方法,标识发生了一次 ISR 集合缩小。

```
def maybeShrinkIsr(replicaMaxLagTimeMs: Long) {
    .....
    replicaManager.isrShrinkRate.mark() // 标识发生了一次 ISR 集合缩小
    .....
}
```

通过 JConsole 可以看到相应的 MBean,如图 4-95 所示。

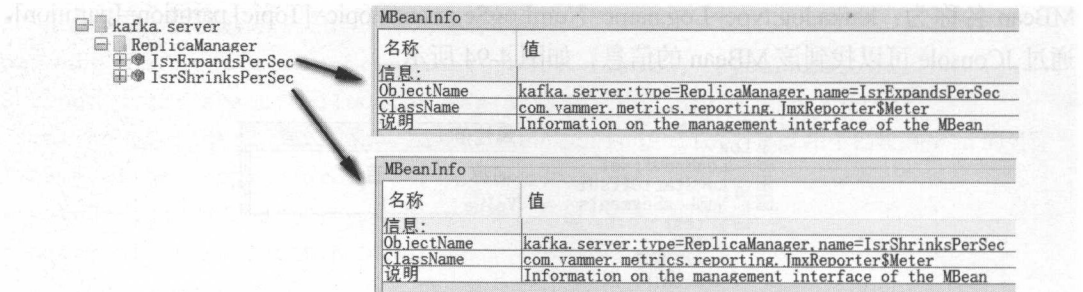


图 4-95

在 `RequestChannel.Request` 中使用 `Histogram` 统计各类请求和响应在 `RequestChannel` 中等待时间的分布。如果有大量请求在 `RequestChannel` 中等待的时间过长，则需要进行调整，例如：`Handler` 线程配置过少，`Kafka` 上下游的服务出现请求洪泛等都会导致问题。观察响应在 `RequestChannel` 中的等待时间也会得到一些类似的结论，相关的代码片段如下：

```
object RequestMetrics {
    val metricsMap = new scala.collection.mutable.HashMap[String, RequestMetrics]
    val consumerFetchMetricName = ApiKeys.FETCH.name + "Consumer"
    val followFetchMetricName = ApiKeys.FETCH.name + "Follower"

    // 为每种类型的请求创建对应的 RequestMetrics 对象
    (ApiKeys.values().toList.map(e => e.name)
    ++ List(consumerFetchMetricName, followFetchMetricName)).foreach(name =>
        metricsMap.put(name, new RequestMetrics(name)))
}

class RequestMetrics(name: String) extends KafkaMetricsGroup {
    val tags = Map("request" -> name)

    // requestQueueTimeHist 负责统计 Request 在 RequestChannel 的等待时间
    val requestQueueTimeHist = newHistogram("RequestQueueTimeMs", biased = true, tags)

    // responseQueueTimeHist 负责统计 Response 在 RequestChannel 中等待的时间
    val responseQueueTimeHist = newHistogram("ResponseQueueTimeMs", biased = true, tags)
```

```
// 这里以上面两个 Histogram 进行介绍, 下面的 Histogram 类似, 请读者了解
// localTimeHist 负责统计 Request 在当前 Broker 中处理的用时
val localTimeHist = newHistogram("LocalTimeMs", biased = true, tags)
// remoteTimeHist 负责统计此 Broker 发送的 Request 在远端 Broker 中处理的用时, 例如
// FetchRequest
val remoteTimeHist = newHistogram("RemoteTimeMs", biased = true, tags)
// responseSendTimeHist 负责统计发送 Response 的用时
val responseSendTimeHist = newHistogram("ResponseSendTimeMs", biased =
true, tags)
.....
}
```

下面介绍 requestQueueTimeHist 和 responseQueueTimeHist 使用过程中涉及的几个时间戳的设置时机, 如图 4-96 所示。

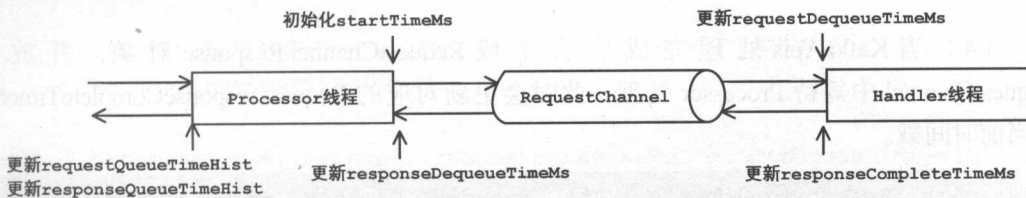


图 4-96

(1) 当在 Processor 线程中成功接收到一个请求时会创建 RequestChannel.Request 对象, 并将 startTimeMs 初始化为当前时间, 将 requestDequeueTimeMs 字段初始化为 -1。

```
case class Request(processor: Int, connectionId: String, session: Session,
private var buffer: ByteBuffer,
startTimeMs: Long, securityProtocol: SecurityProtocol) {
// 下面是一些记录操作时间的字段。注意, 由于这些字段在多个线程中比较和修改, 使用
// @volatile 修饰, 保证可见性
@volatile var requestDequeueTimeMs = -1L
@volatile var responseCompleteTimeMs = -1L
@volatile var responseDequeueTimeMs = -1L
.....
}
```

(2) 之后将请求放入 RequestChannel 中等待 Handler 线程处理。

(3) 当 Handler 线程从 RequestChannel 中取出 Request 对象时会更新其

requestDequeueTimeMs 字段为当前时间戳，再将请求交给 KafkaApis 处理。

```
class KafkaRequestHandler(...) extends Runnable with Logging {
  def run() {
    while (true) {
      .....
      // 从 RequestChannel 中读取 Request
      req = requestChannel.receiveRequest(300)

      // 更新 requestDequeueTimeMs
      req.requestDequeueTimeMs = SystemTime.milliseconds
      apis.handle(req) // 将请求交由 KafkaApis 处理
    }
  }
}
```

(4) 当 KafkaApis 处理完成后会生成 RequestChannel.Response 对象，并放入 RequestChannel 中等待 Processor 处理。此时会更新对应的 Request.responseCompleteTimeMs 为当前时间戳。

```
case class Response(processor: Int, request: Request, responseSend: Send,
  responseAction: ResponseAction) {
  request.responseCompleteTimeMs = SystemTime.milliseconds
  .....
}
```

(5) 之后，Processor 线程从 RequestChannel 中获取 Response 时会更新对应 Request 的 responseDequeueTimeMs 字段为当前时间戳。然后 Processor 线程开始发送 Response 到远端。

```
def receiveResponse(processor: Int): RequestChannel.Response = {
  .....
  response.request.responseDequeueTimeMs = SystemTime.milliseconds
}
```

(6) 当 Processor 发送完成后会调用 Request.updateRequestMetrics() 方法，更新 requestQueueTimeHist 和 responseQueueTimeHist。


```

private def processCompletedSends() {
  selector.completedSends.asScala.foreach { send =>
    .....
    // 调用 updateRequestMetrics() 方法更新度量对象
    resp.request.updateRequestMetrics()
    .....
  }
}

// 下面是 RequestChannel.Request.updateRequestMetrics() 方法的实现
def updateRequestMetrics() {
  .....
  // 计算 Request 在 RequestChannel 中的等待时间
  val requestQueueTime = math.max(requestDequeueTimeMs - startTimeMs, 0)
  // 计算 Response 在 RequestChannel 中的等待时间
  val responseQueueTime = math.max(responseDequeueTimeMs -
    responseCompleteTimeMs, 0)
  .....
  val metricNames = fetchMetricNames :+ ApiKeys.forId(requestId).name
  metricNames.foreach { metricName =>
    val m = RequestMetrics.metricsMap(metricName)

    // 更新 requestQueueTimeHist
    m.requestQueueTimeHist.update(requestQueueTime)
    .....

    // 更新 responseQueueTimeHist
    m.responseQueueTimeHist.update(responseQueueTime)
  }
}
}

```

在 Kafka 中使用 Timer 来统计 Broker 发生增减时, Leader 副本选举的时长及其分布, 更确切地说, 是统计 Controller Leader 执行整个 `BrokerChangeListener.handleChildChange()` 方法所用的时间及其分布。KafkaServer 是服务端的入口, 它启动时会调用 `registerStats()` 方法创建并注册 Timer 对象。


```
// 服务端启动时会调用 KafkaServer.startup() 方法，其中会调用 registerStats() 方法
private def registerStats() {
    BrokerTopicStats.getBrokerAllTopicsStats()
    ControllerStats.uncleanLeaderElectionRate
    ControllerStats.leaderElectionTimer
}

object ControllerStats extends KafkaMetricsGroup {
    ..... // 调用 newTimer 创建 Timer 对象
    private val _leaderElectionTimer = new KafkaTimer(
        newTimer("LeaderElectionRateAndTimeMs", TimeUnit.MILLISECONDS,
        TimeUnit.SECONDS))
    .....
    def leaderElectionTimer: KafkaTimer = _leaderElectionTimer
}
```

其中 `KafkaTimer` 是对 `Timer` 的简单封装，底层还是调用 `Timer.time()` 和 `stop()` 方法实现统计功能，具体实现代码就不贴出来了。

下面来看一下 `BrokerChangeListener` 中对 `leaderElectionTimer` 的使用。

```
class BrokerChangeListener() extends IZkChildListener with Logging {

    def handleChildChange(parentPath: String, currentBrokerList:
    List[String]) {
        inLock(controllerContext.controllerLock) {
            if (hasStarted.get) { // 检测 ReplicaStateMachine 是否已经开启
                ControllerStats.leaderElectionTimer.time {
                    // BrokerChangeListener 中的代码之前已经分析过了，这里就不再贴出来了
                    .....
                }
            }
        }
    }
}
```

4.9.4 Kafka 的监控功能

除了使用 Metrics 工具包对内部状态监控, Kafka 自身还提供了监控模块的实现, 该监控模块的代码在 clients 包下的 org.apache.kafka.common.metrics 以及其子包中, 本小节主要对该监控模块的实现进行分析。首先来分析用于基本的度量类型接口以及实现, 类图如图 4-97 所示。

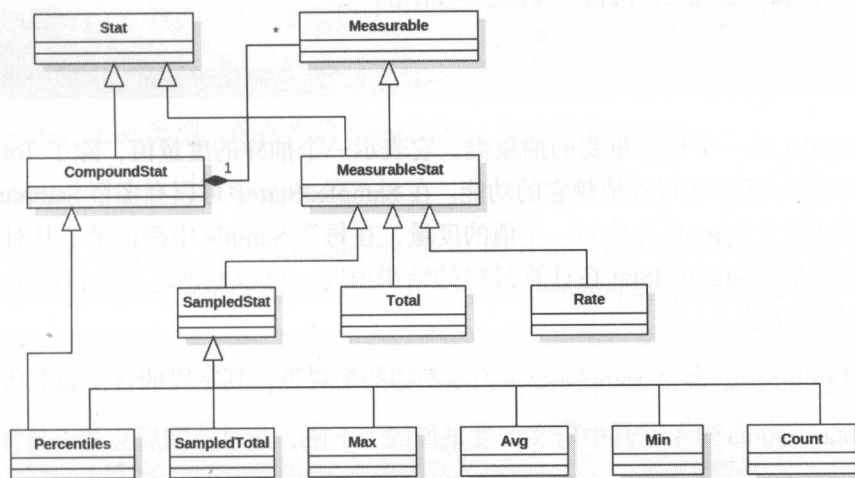


图 4-97

Measurable 接口是度量类型最基础的接口, 通过 measure() 方法获取被监控的值。

```

public interface Measurable {
    public double measure(MetricConfig config, long now);
}
  
```

Stat 接口表示需要经过统计计算的度量类型, 例如平均值、最大值、最小值等, 它通过 record() 方法记录某值并更新度量值。

```

public interface Stat {
    public void record(MetricConfig config, double value, long timeMs);
}
  
```

MeasurableStat 继承了 Measurable 接口和 Stat 接口, 其中并未添加新方法。CompoundStat 接口表示多个 Stat 的组合。

```
public interface CompoundStat extends Stat {
    public List<NamedMeasurable> stats();// 由多个 NamedMeasurable 构成

    public static class NamedMeasurable {
        private final MetricName name;
        private final Measurable stat; // 封装了一个个 Measurable 对象
        ..... // getterr 方法和构造函数省略
    }
}
```

SampledStat 是一个比较重要的抽象类，它表示一个抽样的度量值，除了 Total 外的其他 MeasurableStat 接口实现都依赖它的功能。在 SampledStat 中可以有多多个 Sample（抽样，样本）并通过多个 Sample 完成对一个值的度量，在每个 Sample 中都记录了其对应的时间窗口和事件数量，SampledStat 在计算最终的结果值时，可以根据这两个值决定是否使用此 Sample 中的数据。

MetricConfig 是一个与 SampledStat 关系密切的配置类，其字段的含义如下所述。

- quota: Quota 对象，其中定义了度量值的上下限，超过范围后会抛出异常。
- samples: 指定样本的个数，默认值是 2。
- eventWindow: 指定每个样本中事件的上限，当一个样本中的事件数超过此值后，则开始下个样本的记录，默认值是 Long.MAX_VALUE。
- timeWindowMs: 指定每个样本取样的时间窗口，当一个样本的时间窗口超出范围后，则开始下个样本的记录，默认值是 30 秒。
- tags: Map<String, String> 类型，指定相关的 Tag 标签。

MetricConfig 的方法主要是完成上述字段的读写，这里不再赘述。

Sample 中的各个字段的含义如下所述。

- initialValue: 指定样本的初始值。
- value: 记录样本的值。
- eventCount: 记录当前样本的事件数。
- lastWindowMs: 记录当前样本的时间窗口开始的时间戳。

Sample.isCpmlete() 方法会通过检测 eventCount 和 lastWindows 决定当前样本是否已经

取样完成。

```
public boolean isComplete(long timeMs, MetricConfig config) {
    return timeMs - lastWindowMs >= config.timeWindowMs() // 检测时间窗口
        || eventCount >= config.eventWindow(); // 检测事件数
}
```

SampledStat 的各个字段含义如下所述。

- initialValue: 指定每个样本的初始值。
- samples: List 类型, 保存当前 SampledStat 中的多个 Sample。
- current: 当前使用的 Sample 的下标。

SampledStat 中实现了 MeasurableStat 接口的 record() 方法和 measure() 方法。在 record() 方法中会根据时间窗口和事件数使用合适的 Sample 对象进行记录, 具体代码如下:

```
public void record(MetricConfig config, double value, long timeMs) {
    Sample sample = current(timeMs); // 得到当前的 Sample 对象
    if (sample.isComplete(timeMs, config)) // 检测当前 Sample 是否已完成取样
        sample = advance(config, timeMs); // 获取下一个 Sample
    // 更新 Sample, update() 方法是抽象方法
    update(sample, config, value, timeMs);
    sample.eventCount += 1; // 增加事件数
}
```

// 根据配置指定的 Sample 数量决定创建新 Sample 还是使用之前的 Sample 对象

```
private Sample advance(MetricConfig config, long timeMs) {
    this.current = (this.current + 1) % config.samples();
    if (this.current >= samples.size()) { // 创建新 Sample 对象
        Sample sample = newSample(timeMs);
        this.samples.add(sample);
        return sample;
    } else {
        Sample sample = current(timeMs);
        sample.reset(timeMs); // 重用之前的 Sample 对象
        return sample;
    }
}
```


在 `SampledStat.measure()` 方法中, 首先会将过期的 `Sample` 重置, 之后调用 `combine()` 这个抽象方法完成计算。

```
public double measure(MetricConfig config, long now) {
    purgeObsoleteSamples(config, now);
    return combine(this.samples, config, now);
}

protected void purgeObsoleteSamples(MetricConfig config, long now) {
    long expireAge = config.samples() * config.timeWindowMs(); // 计算过期时长
    for (int i = 0; i < samples.size(); i++) {
        Sample sample = this.samples.get(i);
        if (now - sample.lastWindowMs >= expireAge)
            sample.reset(now); // 检测到 Sample 过期, 则将其重置
    }
}
```

`Total` 是 `MeasurableStat` 接口最简单的实现, 用于记录总数, 在 `record()` 方法中完成累加, 由 `measure()` 方法返回累加值。`Count` 继承了 `SampledStat`, 在 `update()` 方法中完成了 `Sample` 的 `value` 的累加, 在 `combine()` 方法中将所有未过期的 `Sample` 的 `value` 求和并返回。`Count` 与 `Total` 的不同之处在于, 随着时间窗口的推移, `Count` 中会有 `Sample` 过期, 所以 `Count` 记录的是一段时间内的总数, 而 `Total` 的值则单调递增。

`Rate` 实现了 `MeasurableStat` 接口, 它用于记录比率, 例如每秒钟创建连接的个数。`Rate` 中封装了一个 `SampledStat` 类型的对象, 默认是 `Rate` 的内部类 `SampledTotal` 的对象, `SampledTotal` 的实现与 `Count` 类似。`Rate.record()` 方法会调用其中的 `stat` 字段的 `record()` 方法进行记录。

```
public void record(MetricConfig config, double value, long timeMs) {
    // 委托给其中封装的 SampledStat 对象
    this.stat.record(config, value, timeMs);
}
```

`Rate.measure()` 方法会获取其 `stat` 字段的记录值并进行相应计算得到最终的比率。

```

public double measure(MetricConfig config, long now) {
    double value = stat.measure(config, now); // 获取 stat 的记录值
    // 记录值除以总时间, 得到最终的比率。convert() 方法是根据 Rate 设置的时间单位, 对总
    // 时间进行转换, 代码比较简单, 不再赘述
    return value / convert(windowSize(config, now));
}

public long windowSize(MetricConfig config, long now) {
    // 首先, 重置所有已经过期的 Sample, 它们不参与总时间的计算
    stat.purgeObsoleteSamples(config, now);
    // 通过 SampledStat.oldest() 方法获取最旧的 Sample 对象, 从而计算总时间
    long totalElapsedTimeMs = now - stat.oldest(now).lastWindowMs;
    // 有多少个完整的时间窗口, 即有多少个完成的 Sample
    int numFullWindows = (int) (totalElapsedTimeMs / config.timeWindowMs());
    int minFullWindows = config.samples() - 1; // 计算最小要求的完整窗口
    if (numFullWindows < minFullWindows) // 对总时间进行补偿
        totalElapsedTimeMs += (minFullWindows - numFullWindows) * config.
timeWindowMs();
    return totalElapsedTimeMs;
}

```

其他的 MeasurableStat 实现留给读者分析, 这里只简单介绍其含义: Max 记录最大值, Avg 记录平均值, Min 记录最小值, Count 记录个数, Percentiles 记录多个百分比的值。

Kafka 中通过 KafkaMetrics 对上述度量类型做了又一次封装, 在 KafkaMetrics 还封装了 MetricsName 对象和 MetricsConfig 对象。KafkaMetric 所依赖的类如图 4-98 所示。在 KafkaMetric 中还提供了 value() 方法用于获取 Measurable 中的度量值。

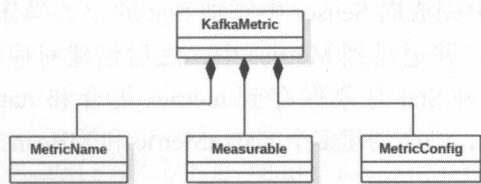


图 4-98

在 MetricsName 中各个字段的含义如下所述。

- name: 记录了 KafkaMetric 的名称。

- **group**: 记录了所属的逻辑组的名称。
- **description**: 保存一些对 `KafkaMetric` 的描述信息。
- **tags**: `Map` 类型, 记录了一些额外的键值对信息, 其中 `group` 和 `tags` 字段用构成 `MBean` 的一部分。

在实际应用中, 对同一个操作需要有多个不同方面的度量, 例如, 我们需要监控请求的最大长度, 也需要监控请求的平均值等。为了满足这种需求, `Kafka` 将多个相关的度量对象封装进 `Sensor` 中。`Sensor` 中的核心字段如下所述。

- **registry**: `Metrics` 对象, 在后面详细介绍。
- **name**: 当前 `Sensor` 对象的名称, `Metrics` 通过该名称区别不同的 `Sensor` 对象。
- **parents**: `Sensor` 数组类型, `Sensor` 是可以分为多层的, 此字段指定了当前 `Sensor` 对象的父 `Sensor`。
- **stats**: `List<Stat>` 类型, 保存构成 `Sensor` 的度量对象。
- **metrics**: `List<KafkaMetric>` 类型, 保存构成 `Sensor` 的 `KafkaMetric` 对象, 当度量对象添加进 `Sensor` 时会创建对应的 `KafkaMetric` 对象 (`CompoundStat` 会创建多个), 并保存到此集合中。
- **config**: `MetricConfig` 类型, 默认的配置信息。
- **lastRecordTime**: 最后一次执行 `record()` 方法的时间戳。
- **inactiveSensorExpirationTimeMs**: 长时间未使用 `Sensor` 会被认为是“过期 `Sensor`”, 由 `ExpireSensorTask` 线程负责进行清理, 此字段记录成为“过期 `Sensor`”的阈值。

在 `Sensor.add()` 方法中完成向 `Sensor` 中添加 `Stat` 的相关操作。它首先会将 `Stat` 对象封装成 `KafkaMetric` 对象, 并记录到 `Metrics` 中, 之后创建对应的 `MBean` 并进行注册, 最后将 `KafkaMetric` 对象和 `Stat` 对象保存到 `metrics` 集合和 `stats` 集合。如果需要添加 `CompoundStat` 类型的对象, 则会创建多个 `KafkaMetric` 和 `MBean` 对象。

```

public synchronized void add(MetricName metricName, MeasurableStat stat,
MetricConfig config) {
    // 创建 KafkaMetric 对象
    KafkaMetric metric = new KafkaMetric(new Object(), Utils.
notNull(metricName),
        Utils.notNull(stat), config == null ? this.config : config, time);
    // 将 KafkaMetric 保存到 Metrics 中, 创建并注册对应 Mbean
    this.registry.registerMetric(metric);
    this.metrics.add(metric); // 添加到 metrics 集合
    this.stats.add(stat); // 添加到 stats 集合
}

public synchronized void add(CompoundStat stat, MetricConfig config) {
    this.stats.add(Utils.notNull(stat));
    // 遍历 CompoundStat 中的每个子 Stat 对象, 处理同上
    for (NamedMeasurable m : stat.stats()) {
        KafkaMetric metric = new KafkaMetric(this, m.name(), m.stat(),
            config == null ? this.config : config, time);
        this.registry.registerMetric(metric);
        this.metrics.add(metric);
    }
}
}

```

在 `Sensor.record()` 方法中完成了记录功能。它会更新 `lastRecordTime` 字段, 调用 `stats` 集合中所有 `Stat` 对象以及 `parents` 集合中所有父 `Sensor` 的 `record()` 方法。

```

public void record(double value, long timeMs) {
    this.lastRecordTime = timeMs;
    synchronized (this) {
        // 调用 stats 集合中每个 Stat 对象的 record() 方法
        for (int i = 0; i < this.stats.size(); i++)
            this.stats.get(i).record(config, value, timeMs);
        checkQuotas(timeMs); // 检测是否超出了 MetricConfig 指定的上下限
    }
}

```



```

    }
    for (int i = 0; i < parents.length; i++)
        parents[i].record(value, timeMs); // 调用每个父 Sensor 的 record() 方法
    }

    private void checkQuotas(long timeMs) {
        for (int i = 0; i < this.metrics.size(); i++) {
            KafkaMetric metric = this.metrics.get(i);
            MetricConfig config = metric.config(); // 获取 MetricsConfig 对象
            if (config != null) {
                Quota quota = config.quota(); // 获取 MetricsConfig 对象中的 Quota 对象
                if (quota != null) {
                    double value = metric.value(timeMs); // 计算最终的度量值
                    if (!quota.acceptable(value)) { // 检测度量值是否超出上下限
                        throw new QuotaViolationException(String.format("..."));
                    }
                }
            }
        }
    }
}

```

下面我们来分析 Metrics 类，它负责统一管理 Sensor 对象、KafkaMetric 对象以及 MetricsReporter 对象，如图 4-99 所示。

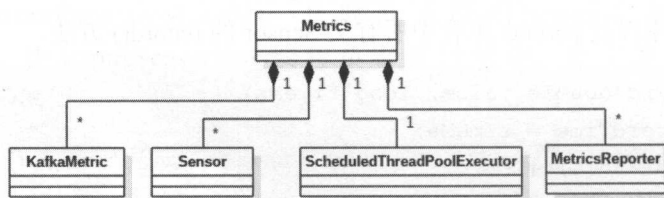


图 4-99

Metrics 中各个字段的含义和功能如下所述。

- config: MetricConfig 对象，默认的配置信息。
- metrics: ConcurrentMap<MetricName, KafkaMetric> 类型，保存了添加到 Metrics 中的 KafkaMetric 对象。

- `sensors`: `ConcurrentMap<String, Sensor>` 类型, 保存了添加到 `Metrics` 中的 `Sensor` 对象。
- `childrenSensors`: 记录了每个 `Sensor` 的子 `Sensor` 集合。
- `reporters`: `List<MetricsReporter>` 类型, 保存了使用的 `MetricsReporter` 对象, 默认是 `JmxReporter`。
- `metricsScheduler`: JDK 提供的 `ScheduledThreadPoolExecutor` 类型, 用于执行 `ExpireSensorTask` 定时任务。

`Metrics.sensor()` 方法主要负责从 `sensors` 集合中获取 `Sensor` 对象, 如果指定的 `Sensor` 不存在则创建新 `Sensor` 对象, 并使用 `childrenSensors` 集合记录 `Sensor` 的层级关系。

```
public synchronized Sensor sensor(String name, MetricConfig config,
    long inactiveSensorExpirationTimeSeconds, Sensor... parents) {
    Sensor s = getSensor(name); // 根据 name 从 sensors 集合中获取 Sensor 对象
    if (s == null) {
        s = new Sensor(this, name, parents, config == null ? this.config :
config,
            // 创建 Sensor 对象
            time, inactiveSensorExpirationTimeSeconds);
        this.sensors.put(name, s); // 保存到 sensors 集合中
        if (parents != null) {
            // 通过 childrenSensors 记录 Sensor 的层次关系
            for (Sensor parent : parents) {
                List<Sensor> children = childrenSensors.get(parent);
                if (children == null) {
                    children = new ArrayList<>();
                    childrenSensors.put(parent, children);
                }
                children.add(s);
            }
        }
    }
    return s;
}
```

`Metrics.addReporter()` 方法负责添加 `MetricsReporter`, 在通过 `addMetric()` 方法添加 `KafkaMetric` 对象时会调用 `registerMetric()` 方法向每个 `MetricReporter` 进行注册。

```

synchronized void registerMetric(KafkaMetric metric) {
    MetricName metricName = metric.metricName();
    if (this.metrics.containsKey(metricName))
        throw new IllegalArgumentException("...");
    this.metrics.put(metricName, metric); // 向 metrics 中添加 KafkaMetric 对象
    for (MetricsReporter reporter : reporters)
        // 向每个 MetricsReporter 中注册 KafkaMetric 对象
        reporter.metricChange(metric);
}

```

在 Metrics 的构造方法中除了初始化其字段，还会通过 metricsScheduler 启动一个 ExpireSensorTask 定时任务处理“过期 Sensor”，创建一个用于记录 metrics 集合大小的匿名度量类的对象。

```

public Metrics(MetricConfig defaultConfig, List<MetricsReporter> reporters,
    Time time, boolean enableExpiration) {
    ..... // 初始化其他字段
    // 创建 ScheduledThreadPoolExecutor
    if (enableExpiration) {
        this.metricsScheduler = new ScheduledThreadPoolExecutor(1);
        this.metricsScheduler.setThreadFactory(new ThreadFactory() {
            public Thread newThread(Runnable runnable) {
                return Utils.newThread("SensorExpiryThread", runnable, true);
            }
        });
        this.metricsScheduler.scheduleAtFixedRate(new ExpireSensorTask(),
            30, 30, TimeUnit.SECONDS); // 启动定时任务
    } else {
        this.metricsScheduler = null;
    }

    addMetric(metricName("count", "kafka-metrics-count",
        "total number of registered metrics"),
        // 创建一个记录 metrics 集合大小的 Measurable 对象，并注册到 Metrics 中
        new Measurable() {
            public double measure(MetricConfig config, long now) {
                return metrics.size();
            }
        });
}

```


在 `ExpireSensorTask` 任务中会遍历 `sensors` 集合，并将过期的 `Sensor` 对象删除，代码比较简单，就不贴出来了。

在 `Metrics.reporters` 集合中可以保存多个 `MetricsReporter` 对象，当添加 `KafkaMetrics` 时会调用 `MetricReporter.metricChange()` 方法进行注册，`MetricReporter` 会按照其实现的方式将 `KafkaMetrics` 中的度量信息暴露出来，例如 `JmxReporter` 实现会创建并注册相应的 `MBean`。`MetricsReporter` 接口中定义了如下方法：

```
public interface MetricsReporter extends Configurable {
    // 初始化函数，将 MetricsReporter 添加到 Metrics 时被调用，完成初始化操作
    public void init(List<KafkaMetric> metrics);
    // 向 Metrics 中添加 KafkaMetric 时被调用
    public void metricChange(KafkaMetric metric);
    // 从 Metrics 中删除 KafkaMetric 时被调用
    public void metricRemoval(KafkaMetric metric);
    public void close(); // 关闭操作
}
```

`JmxReporter` 是 Kafka 提供的 `MetricsReporter` 接口默认实现类，它通过 JMX 的方式将 `KafkaMetric` 中的信息暴露出来，它使用 `mbeans` 集合（`Map<String, KafkaMBean>`）记录了 `MBean` 的名称和对应的 `KafkaMBean` 对象。`KafkaMBean` 是一个 `DynamicMBean` 接口的实现，其中通过 `metrics`（`Map<String, KafkaMetric>`）集合记录了添加的 `KafkaMetric` 对象，这些 `KafkaMetric` 对象会被当作 `KafkaMBean` 对象的属性处理。

```
private static class KafkaMBean implements DynamicMBean {
    private final ObjectName objectName; // MBean 的名称
    // 保存了添加的 KafkaMetric 对象
    private final Map<String, KafkaMetric> metrics;
    ..... // 构造函数会初始化 objectName 和 metrics 这两个字段，不再赘述

    public void setAttribute(String name, KafkaMetric metric) {
        this.metrics.put(name, metric); // 记录 KafkaMetric 对象
    }

    @Override
    public Object getAttribute(String name) throws AttributeNotFoundException,
        MBeanException, ReflectionException {
        if (this.metrics.containsKey(name))
```



```

        // 返回指定 KafkaMetric 中的度量值
        return this.metrics.get(name).value();
    }
    else
        throw new AttributeNotFoundException("Could not find attribute
" + name);
    }

    @Override
    public AttributeList getAttributes(String[] names) {
        // 循环遍历 name 并调用 getAttribute() 方法
    }

    public KafkaMetric removeAttribute(String name) {
        return this.metrics.remove(name);
    }
}
// 继承自 DynamicMBean 接口的 invoke() 方法、setAttribute() 方法、setAttributes() 方法
// 具体实现就是直接抛出 UnsupportedOperationException 异常
}

```

JmxReporter.metricChange() 方法会将 KafkaMetric 以属性的形式添加到 KafkaMBean 中，并重新注册 KafkaMBean。

```

public void metricChange(KafkaMetric metric) {
    synchronized (LOCK) {
        KafkaMBean mbean = addAttribute(metric); // 添加 KafkaMetric
        reregister(mbean); // 重新注册 KafkaMBean
    }
}

private KafkaMBean addAttribute(KafkaMetric metric) {
    try {
        MetricName metricName = metric.metricName();
        String mbeanName = getMBeanName(metricName); // 获取 KafkaMBean 的名称
        // 如果没有此名称的 KafkaMBean 则创建
        if (!this.mbeans.containsKey(mbeanName))
            mbeans.put(mbeanName, new KafkaMBean(mbeanName));
        KafkaMBean mbean = this.mbeans.get(mbeanName);
    }
}

```

```

        mbean.setAttribute(metricName.name(), metric); // 以属性的形式添加
        KafkaMetric
        return mbean;
    } catch (JMException e) {
        throw new KafkaException("...");
    }
}

private void reregister(KafkaMBean mbean) {
    unregister(mbean); // 先取消 KafkaMBean 的注册
    try {
        // 调用 MBeanServer.registerMBean() 方法重新注册 KafkaMBean
        ManagementFactory.getPlatformMBeanServer().registerMBean(mbean,
        mbean.name());
    } catch (JMException e) {
        throw new KafkaException("...");
    }
}
}

```

最后注意一下 `getMBeanName()` 方法生成的 `KafkaMBean` 名称的格式是 “`prefix:type=[group],key1=value1,key2=value2...`”。

```

private String getMBeanName(MetricName metricName) {
    StringBuilder mBeanName = new StringBuilder();
    mBeanName.append(prefix); // 第一部分是前缀，服务端默认是 kafka.server
    mBeanName.append(":type="); // 第二部分是 MetricName 中的 group 字段
    mBeanName.append(metricName.group());
    // 第三部分由 MetricName 中的 tags 集合构成
    for (Map.Entry<String, String> entry : metricName.tags().entrySet()) {
        if (entry.getKey().length() <= 0 || entry.getValue().length() <= 0)
            continue;
        mBeanName.append(",");
        mBeanName.append(entry.getKey());
        mBeanName.append("=");
        mBeanName.append(entry.getValue());
    }
    return mBeanName.toString();
}
}

```

Kafka 自带的监控模块的相关实现和原理到这里就介绍完了。在下一小节将以监控 KSelector 相关指标为例介绍该模块的使用。

4.9.5 监控 KSelector 的指标

在前面分析 KafkaController 时知道，在 Controller 成为 Leader 时会创建 ControllerChannelManager，它负责管理 Controller Leader 与其他 Broker 之间的连接。涉及网络连接的问题都会与 NetworkClient 和 KSelector 相关，本节就以 ControllerChannelManager 中使用的 KSelector 为例，分析 Kafka 监控模块的使用。

读者可以回顾前面对 ControllerChannelManager.addNewBroker() 方法的介绍，其中会为每个 Broker 创建 RequestSendThread 线程、对应的 NetworkClient 对象、消息队里等，这里只关注 KSelector 的构造函数。

```
private def addNewBroker(broker: Broker) {  
    ..... // 创建消息队列 (略)  
    val networkClient = {  
        ..... // 创建一个 ChannelBuilder (略)  
        ..... // 创建 Selector 对象  
        val selector = new Selector(NetworkReceive.UNLIMITED, config.  
connectionsMaxIdleMs,  
        metrics, time, "controller-channel",  
        Map("broker-id" -> broker.id.toString).asJava,  
        false, channelBuilder  
    )  
    ..... // 创建 NetworkClient (略)  
    }  
    ..... // 创建一个 RequestSendThread 线程 (略)  
}
```

注意 KSelector 构造方法的第三个参数，它是一个 Metrics 对象，在 KafkaServer 启动时创建，具体代码片段如下：


```

var metrics: Metrics = null
private val metricConfig: MetricConfig = new MetricConfig()
    .samples(config.metricNumSamples) // 设置 Sample 的个数, 默认是 2
    // 设置 Sample 的时间
    .timeWindow(config.metricSampleWindowMs, TimeUnit.MILLISECONDS)

// JMXPrefix 是用于构成 MBean 名称的一部分
private val jmxPrefix: String = "kafka.server"

// 读取配置的 MetricReporter 类型
private val reporters: java.util.List[MetricReporter] = config.
metricReporterClasses

reporters.add(new JmxReporter(jmxPrefix)) // 默认会创建添加 JmxReporter 对象

// 创建 Metrics 对象, 其中第四个参数是 enableExpiration,
// 即标识开启 ExpireSensorTask 任务
metrics = new Metrics(metricConfig, reporters, kafkaMetricsTime, true)

```

回到 KSelector 继续分析, 在 KSelector 中与监控相关的字段如下所述。

- metricTags: 创建 MetricName 时使用的 tags 集合, 会成为 MBean 名称的一部分。
- metricGrpPrefix: MetricName 中 group 的前缀, 此处为 “controller-channel”。
- sensors: SelectorMetrics 对象, 其中封装了 KSelector 中使用到的全部 Sensor 对象。

SelectorMetrics 是 KSelector 中的私有内部类, 其中封装了多个 Sensor 对象。SelectorMetrics 中各个字段的含义如下所述。

- connectionClosed: 监控连接关闭, 使用 Rate 记录每秒连接关闭数。
- connectionCreated: 监控连接创建, 使用 Rate 记录每秒连接创建数
- bytesTransferred: 监控网络操作数, 使用 Rate 记录每秒钟所有连接上执行的读写操作总数。
- bytesSent: 监控发送请求的相关指标, 此 Sensor 是 bytesTransferred 的子 Sensor。其中使用 Rate 记录 Controller 每秒钟发送的字节数和请求数, 使用 Avg 记录发送请求的平均大小, 使用 Max 记录发送请求的最大长度。

- **bytesReceived**: 监控接收请求的相关指标, 此 Sensor 是 `byteTransferred` 的子 Sensor。使用 `Rate` 记录 Controller 每秒钟接收的字节数和请求数。
- **selectTime**: 监控 `select()` 方法的相关指标。使用 `Rate` 记录每秒钟调用 `select()` 方法的次数, 使用 `Avg` 记录调用 `select()` 方法阻塞的平均值, 使用 `Rate` 记录调用 `select()` 方法阻塞时间占总时间的比例。
- **ioTime**: 监控 I/O 耗时的相关指标。使用 `Avg` 记录 I/O 操作的平均耗时, 使用 `Rate` 执行 I/O 操作占总时间的比例。
- **sensors**: `List<Sensor>` 集合, 保存了上述全部 Sensor 对象。
- **topLevelMetricNames**: `List<MetricName>` 集合, 保存了直接向 Metrics 注册的 `Measurable` 对象, 这些 `Measurable` 对象没有注册到 Sensor 中。

在 `SensorMetrics` 的构造函数中会创建上述 Sensor 对象, 并按照上面描述的功能添加 `MeasurableStat`, 此方法代码有些长, 请读者耐心分析。

```
public SelectorMetrics(Metrics metrics) {
    this.metrics = metrics;
    // 此处得到的 metricGrpName 的值为 "controller-channel-metrics", 下面所有
    // Sensor 都是用此值作为 group 部分
    String metricGrpName = metricGrpPrefix + "-metrics";
    StringBuilder tagsSuffix = new StringBuilder();
    // 将 tags 集合组装成字符串, 假设连接的 BrokerId 为 1, 此值为 "broker-1", 下面所有
    // Sensor 都会将此值作为其 name 的一部分
    for (Map.Entry<String, String> tag : metricTags.entrySet()) {
        tagsSuffix.append(tag.getKey());
        tagsSuffix.append("-");
        tagsSuffix.append(tag.getValue());
    }
    // 创建 Sensor 对象
    this.connectionClosed = sensor("connections-closed:" + tagsSuffix.
toString());
    // 创建 MetricsName
    MetricName metricName = metrics.metricName("connection-close-rate",
        metricGrpName, "Connections closed per second in the window.",
metricTags);
```

```

// 添加 Rate 记录每秒连接的关闭数
this.connectionClosed.add(metricName, new Rate());

this.connectionCreated = sensor("connections-created:" + tagsSuffix.
toString());

metricName = metrics.metricName("connection-creation-rate",
metricGrpName,
    "New connections established per second in the window.",
metricTags);

this.connectionCreated.add(metricName, new Rate()); // 记录每秒连接的创建数

this.bytesTransferred = sensor("bytes-sent-received:" + tagsSuffix.
toString());

metricName = metrics.metricName("network-io-rate", metricGrpName,
    "...", metricTags);
// 记录所有连接每秒执行的读写总数
bytesTransferred.add(metricName, new Rate(new Count()));

this.bytesSent = sensor("bytes-sent:" + tagsSuffix.toString(),
    bytesTransferred); // 指定 bytesTransferred 为父 Sensor
metricName = metrics.metricName("outgoing-byte-rate", metricGrpName,
    "...", metricTags);
this.bytesSent.add(metricName, new Rate()); // 记录所有连接每秒发送的总字节数
metricName = metrics.metricName("request-rate", metricGrpName,
    "The average number of requests sent per second.",
    metricTags);
// 记录平均每秒发送的总请求数
this.bytesSent.add(metricName, new Rate(new Count()));
metricName = metrics.metricName("request-size-avg", metricGrpName,
    "The average size of all requests in the window..", metricTags);
this.bytesSent.add(metricName, new Avg()); // 记录请求平均的平均大小
metricName = metrics.metricName("request-size-max", metricGrpName,
    "The maximum size of any request sent in the window.", metricTags);
this.bytesSent.add(metricName, new Max()); // 记录请求的最大长度

this.bytesReceived = sensor("bytes-received:" + tagsSuffix.toString(),
    bytesTransferred); // 指定 bytesTransferred 为父 Sensor

```

```

metricName = metrics.metricName("incoming-byte-rate", metricGrpName,
    "Bytes/second read off all sockets", metricTags);
this.bytesReceived.add(metricName, new Rate()); // 记录每秒收到的字节数
metricName = metrics.metricName("response-rate", metricGrpName,
    "Responses received sent per second.", metricTags);
// 记录每秒收到的请求数
this.bytesReceived.add(metricName, new Rate(new Count()));

this.selectTime = sensor("select-time:" + tagsSuffix.toString());
metricName = metrics.metricName("select-rate", metricGrpName,
    "...", metricTags);
// 记录每秒调用 select() 方法次数
this.selectTime.add(metricName, new Rate(new Count()));
metricName = metrics.metricName("io-wait-time-ns-avg", metricGrpName,
    "...", metricTags);
// 记录 select() 方法的平均阻塞时间
this.selectTime.add(metricName, new Avg());
metricName = metrics.metricName("io-wait-ratio", metricGrpName,
    "The fraction of time the I/O thread spent waiting.", metricTags);
// 记录调用 select() 方法阻塞时间占总时间的比例
this.selectTime.add(metricName, new Rate(TimeUnit.NANOSECONDS));

this.ioTime = sensor("io-time:" + tagsSuffix.toString());
metricName = metrics.metricName("io-time-ns-avg", metricGrpName,
    "...", metricTags);
this.ioTime.add(metricName, new Avg()); // 记录 I/O 的平均时长
metricName = metrics.metricName("io-ratio", metricGrpName,
    "The fraction of time the I/O thread spent doing I/O",
metricTags);
// 记录执行 I/O 时间占总时间的比例
this.ioTime.add(metricName, new Rate(TimeUnit.NANOSECONDS));

metricName = metrics.metricName("connection-count", metricGrpName,
    "The current number of active connections.", metricTags);
topLevelMetricNames.add(metricName);
// 直接向 Metrics 添加匿名 Measurable, 用来记录连接数
this.metrics.addMetric(metricName, new Measurable() {

```



```

public double measure(MetricConfig config, long now) {
    return channels.size();
}
});
}

```

通过对构造函数的分析可以得到 SelectorMetrics 的结构，如图 4-100 所示。

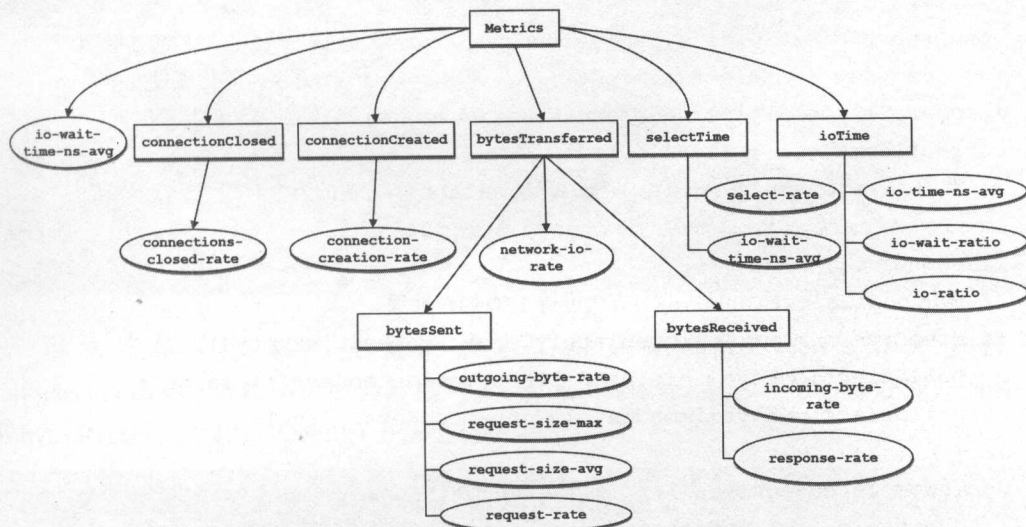


图 4-100

下面通过分析 SelectorMetrics 中每个 Sensor 的 record() 方法调用位置，进一步理解每个 Sensor 的含义。在 KSelect.close() 方法中会关闭指定的连接，同时会调用 connections-closed 字段的 record() 方法进行记录。

```

private void close(KafkaChannel channel) {
    try {
        channel.close(); // 关闭连接
    } catch (IOException e) {
        log.error("Exception closing connection to node {}: ", channel.id(), e);
    }
    .....
    // 调用 connectionClosed.record() 方法，记录连接关闭数
    this.sensors.connectionClosed.record();
}

```


在 `KSelector.poll()` 方法中会通过调用 `KSelector.select()` 方法监听是否有连接触发其关注的事件，其中会计算 `select()` 方法的起止时间并调用 `selectTime` 字段的 `record()` 方法进行记录。`select()` 方法返回后会调用 `KSelector.pollSelectionKeys()` 方法进行 I/O 操作，处理 `select()` 方法得到的连接，其中会记录 I/O 操作的起止时间，并调用 `ioTime` 字段的 `record()` 方法记录 I/O 操作的时间。

```
public void poll(long timeout) throws IOException {
    .....
    long startSelect = time.nanoseconds(); // 记录 select() 方法的起始时间
    int readyKeys = select(timeout); // 调用 select() 方法，等待事件发生
    long endSelect = time.nanoseconds(); // 记录 select() 方法的结束时间
    currentTimeNanos = endSelect;
    // 调用 selectTime.record() 方法，记录 select() 方法的阻塞时间
    this.sensors.selectTime.record(endSelect - startSelect, time.
milliseconds());
    // 调用 pollSelectionKeys() 方法进行 I/O 操作处理
    if (readyKeys > 0 || !immediatelyConnectedKeys.isEmpty()) {
        pollSelectionKeys(this.nioSelector.selectedKeys(), false);
        pollSelectionKeys(immediatelyConnectedKeys, true);
    }
    addToCompletedReceives(); // 处理读取到的消息
    long endIo = time.nanoseconds(); // 记录 I/O 操作的结束时间
    // 调用 ioTime.record() 方法，记录 I/O 的耗时
    this.sensors.ioTime.record(endIo - endSelect, time.milliseconds());
    .....
}
```

当检测到连接创建成功时会调用 `connectionCreated` 字段的 `record()` 方法进行记录；检测到连接可写时，会向连接写入请求，当写入一个完整的请求后，会将请求放入 `completedSends` 集合等待后续处理，同时调用 `bytesSent` 字段的 `record()` 方法进行记录。

```
private void pollSelectionKeys(Iterable<SelectionKey> selectionKeys,
    boolean isImmediatelyConnected) {
    .....
    if (isImmediatelyConnected || key.isConnectable()) {
        if (channel.finishConnect()) { // 检测是否已连接成功
            this.connected.add(channel.id());
            // 调用 connectionCreated.record() 方法，记录连接创建数
```

```

        this.sensors.connectionCreated.record();
    } else
        continue;
}
.....
if (channel.ready() && key.isWritable()) {
    Send send = channel.write(); // 向 Channel 写入请求
    if (send != null) { // 成功发送一个完整的请求
        this.completedSends.add(send); // 添加到 completedSends 集合等待后续处理
        // 调用 bytesSent 的 record() 方法进行记录
        this.sensors.recordBytesSent(channel.id(), send.size());
    }
}
.....
}

```

当从连接中读取到完整的请求后会放入 `stagedReceives` 集合中暂存，`pollSelectionKeys()` 方法结束后会转移到 `completedReceives` 集合等待后续处理，同时会调用 `bytesReceived` 字段的 `record()` 方法进行记录。

```

private void addToCompletedReceives() {
    .....
    this.completedReceives.add(networkReceive);
    // 调用 bytesReceived 的 record() 方法进行记录
    this.sensors.recordBytesReceived(channel.id(), networkReceive.
payload().limit());
    .....
}

```

注意，`bytesSent` 和 `bytesReceived` 这两个 Sensor 是 `bytesTransferred` 的子 Sensor，当调用子 Sensor 的 `record()` 方法时会同时调用父 Sensor 的 `record()` 方法。

KSelector 中涉及的 Sensor 的含义以及使用代码的分析就到这里，我们可以通过 JConsole 查看到相应的 MBean 和对应的统计信息，如图 4-101 所示。

kafka.server		属性值	
controller-channel-metrics		名称	值
属性		connection-close-rate	3.0
connection-creation-rate		connection-count	10.0
response-rate		connection-creation-rate	0.0
select-rate		incoming-byte-rate	341.0
connection-count		io-ratio	10.602223462248357
network-io-rate		io-time-ns-avg	2.0
io-ratio		io-wait-ratio	3.0
io-wait-time-ns-avg		io-wait-time-ns-avg	3.0
io-wait-ratio		network-io-rate	5.0
outgoing-byte-rate		outgoing-byte-rate	9.0
request-size-max		request-rate	0.0643859724428038
io-time-ns-avg		request-size-avg	164.66666666666666
request-rate		request-size-max	228.0
incoming-byte-rate		response-rate	3.0
connection-close-rate		select-rate	5.0124458645132551
request-size-avg			

图 4-101

在实际生产中，除了使用 JConsole，还有 Mx4jLoader、Kafka Web Console、Kafka Manager、KafkaOffsetMonitor 等可视化的工具可供选择，这些工具的搭建和使用比较简单，读者可以参考相关文档学习。

本节首先介绍了 JMX 的目的和三层结构，Standard MBean 和 DynamicMBean 的使用方式以及使用 JConsole 查看 MBean 的方式。然后介绍了 Yammer Metrics 工具包的基本原理和常用度量类的使用，简略分析了其 JmxReporter 的实现。之后分析了 Kafka 中对 Yammer Metric 工具包的封装，Kafka 提供的监控模块的实现。最后以 ControllerChannelManager 中使用的 KSelector 为例分析了 Kafka 的监控模块的使用。

第 5 章

Kafka Tool

为了方便管理和使用，Kafka 提供了很多管理脚本，Linux 版本的管理脚本存放在 `$KAFKA_HOME/bin` 目录下（在 `$KAFKA_HOME/bin/windows` 目录下可以找到对应的 Windows 脚本）。下面先来介绍常用脚本的功能。

- `kafka-server-start` 脚本：启动 Kafka Server。
- `kafka-server-stop` 脚本：停止 Kafka Stop。
- `kafka-topics` 脚本：负责 Topic 相关操作，例如，创建 Topic，查询 Topic 名称以及详细信息，增加分区数量并完成新增分区的副本等。
- `kafka-preferred-replica-election` 脚本：触发指定的分区进行“优先副本”选举，这样可以让分区 Leader 副本在集群中分布得更均匀。
- `kafka-reassign-partitions` 脚本：主要有三个功能，一是生成副本迁移的方案，二是触发副本迁移操作，即将迁移方案写入到 ZooKeeper 中，从而触发 `PartitionsReassignListener` 处理，三是检测指定分区的副本迁移是否已完成。
- `kafka-console-producer` 脚本：控制台版本的生产者，我们可以在控制台中输入消息的 key 和 value，由此脚本封装成消息并发送给服务端。
- `kafka-console-consumer` 脚本：控制台版本的消费者，我们可以通过参数指定订阅的 Topic，此脚本会从服务端拉取消息并输出到控制台。
- `kafka-consumer-groups` 脚本：有两个主要的功能，一是查询当前所有 Consumer Group，二是获取指定 Consumer Group 的详细信息。

- **DumpLogSegments**: 可由 `kafka-run-class` 脚本运行, 主要负责解析输出指定的日志文件和索引文件中的内容, 另外还可以实现索引文件的验证。
- **kafka-producer-perf-test** 脚本: 负责测试生产者的各项性能指标。
- **kafka-consumer-perf-test** 脚本: 负责测试消费者的各项性能指标。
- **kafka-mirror-maker** 脚本: 实现了数据在多个集群的同步, 可用于 Kafka 集群的镜像制作。

本章对上述常用脚本的实现进行分析, 由于篇幅限制, 并不会介绍 `$KAFKA_HOME/bin` 目录下所有脚本的使用和实现, 未介绍的脚本请读者参考相关资料和源码进行学习。

5.1 kafka-server-start 脚本

`kafka-server-start` 脚本通过 `kafka-run-class` 脚本调用 `Kafka` 类来启动 `Broker`, 在调用 `kafka-run-class` 脚本之前会进行检测命令行参数、设置 `log4j` 配置文件、设置 JVM 内存参数等操作。

```
#!/bin/bash
if [ $# -lt 1 ]; # 检查参数个数
then # 打印命令用法
    echo "USAGE: $0 [-daemon] server.properties [--override property=value]"
    exit 1
fi
base_dir=$(dirname $0) # 获取当前脚本所在的路径
# 设置 Log4j 相关的环境变量
if [ "x$KAFKA_LOG4J_OPTS" = "x" ]; then
    export KAFKA_LOG4J_OPTS=
        "-Dlog4j.configuration=file:$base_dir/../../config/log4j.properties"
fi
# 设置 JVM 的内存
if [ "x$KAFKA_HEAP_OPTS" = "x" ]; then
    export KAFKA_HEAP_OPTS="-Xmx1G -Xms1G"
fi

EXTRA_ARGS="-name kafkaServer -loggc"
# 检测第一个参数是否为 "-daemon"
COMMAND=$1
```

```

case $COMMAND in
  -daemon)
    EXTRA_ARGS="-daemon "$EXTRA_ARGS
    shift # 左移参数列表, 即删除 "-daemon" 参数
    ;;
  *)
    ;;
esac
# 调用 kafka-run-class 脚本
exec $base_dir/kafka-run-class.sh $EXTRA_ARGS kafka.Kafka "$@"

```

很多脚本和工具类都依赖于 `kafka-run-class` 脚本, 其中的主要功能是设置 `CLASSPATH`, 进行 `JMX` 的相关配置, 配置 `Log4j`, 指定存放日志文件和索引文件位置, 检测 `JAVA_HOME` 环境变量, 进行 `JVM` 的相关配置, 决定是否后台启动。

```

#!/bin/bash
# ... 检测参数并打印使用方法 (略)
# 检测 INCLUDE_TEST_JARS 变量是否为空
if [ -z "$INCLUDE_TEST_JARS" ]; then
  INCLUDE_TEST_JARS=false
fi

# 下面定义函数 should_include_file, 其功能是检测 CLASSPATH 是否需要包含指定的文件
regex="(-(test|src|scaladoc|javadoc)\.jar|jar.asc)$"
should_include_file() {
  if [ "$INCLUDE_TEST_JARS" = true ]; then
    return 0
  fi
  file=$1
  if [ -z "$(echo "$file" | egrep "$regex")" ]; then
    return 0
  else
    return 1
  fi
}

# 获取脚本所在目录的上一层目录, 即 base_dir 指向了根目录 ($KAFKA_HOME)
base_dir=$(dirname $0)/..
# 检测并设置 SCALA_VERSION、SCALA_BINARY_VERSION (略)

```

```
# 检测 $base_dir 下的多个目录, 根据 should_include_file 函数设置 CLASSPATH (略)
# JMX 的相关设置
if [ -z "$KAFKA_JMX_OPTS" ]; then
    KAFKA_JMX_OPTS="-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false "
fi
# 设置 JMX 的端口
if [ $JMX_PORT ]; then
    KAFKA_JMX_OPTS="$KAFKA_JMX_OPTS -Dcom.sun.management.jmxremote.
port=$JMX_PORT "
fi

# 指定存放日志文件和索引文件的目录, 默认是 $KAFKA_HOME/logs
if [ "x$LOG_DIR" = "x" ]; then
    LOG_DIR="$base_dir/logs"
fi
# ..... Log4j 的相关设置 (略)
# ..... 检测是否以 Debug 模式启动 (略)

# 检测 JAVA_HOME 环境变量
if [ -z "$JAVA_HOME" ]; then
    JAVA="java"
else
    JAVA="$JAVA_HOME/bin/java"
fi

# 配置 JVM 内存
if [ -z "$KAFKA_HEAP_OPTS" ]; then
    KAFKA_HEAP_OPTS="-Xmx256M"
fi

# 对 JVM 进行一些优化配置
if [ -z "$KAFKA_JVM_PERFORMANCE_OPTS" ]; then
    KAFKA_JVM_PERFORMANCE_OPTS=\
"-server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 \
-XX:InitiatingHeapOccupancyPercent=35 \
-XX:+DisableExplicitGC -Djava.awt.headless=true"
```



```

fi
# ..... 处理参数 name、loggc、daemon 三个参数 (略)

# 调整 JVM GC 相关的参数
GC_FILE_SUFFIX='-gc.log'
GC_LOG_FILE_NAME=''
if [ "x$GC_LOG_ENABLED" = "xtrue" ]; then
    GC_LOG_FILE_NAME=$DAEMON_NAME$GC_FILE_SUFFIX
    KAFKA_GC_LOG_OPTS="-Xloggc:$LOG_DIR/$GC_LOG_FILE_NAME \
        -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps \
        -XX:+PrintGCTimeStamps "
fi

# 根据 $DAEMON_MODE 的值, 决定是否后台启动
if [ "x$DAEMON_MODE" = "xtrue" ]; then
    nohup $JAVA $KAFKA_HEAP_OPTS $KAFKA_JVM_PERFORMANCE_OPTS \
        $KAFKA_GC_LOG_OPTS $KAFKA_JMX_OPTS $KAFKA_LOG4J_OPTS \
        -cp $CLASSPATH $KAFKA_OPTS "$@" > "$CONSOLE_OUTPUT_FILE" 2>&1 < /dev/
    null &
else
    exec $JAVA $KAFKA_HEAP_OPTS $KAFKA_JVM_PERFORMANCE_OPTS \
        $KAFKA_GC_LOG_OPTS $KAFKA_JMX_OPTS $KAFKA_LOG4J_OPTS -cp $CLASSPATH
    $KAFKA_OPTS "$@"
fi

```

kafka-server-stop 脚本比较简单, 它直接调用 kill 命令杀死 Kafka 服务端进程, 前面介绍过在 Kafka 启动时注册了 JVM 关闭钩子, 此时会调用 KafkaServer.shutdown() 方法关闭所有相关的组件, 不再赘述了。

5.2 kafka-topics 脚本

kafka-topics 脚本主要负责 Topic 相关的操作。它的具体实现是通过上面分析的 kafka-run-class 来直接调用 TopicCommand 类, 并根据参数执行指定的功能。

TopicCommand.main() 方法是该脚本的入口函数, 其中会使用 joptsimple 命令行解释器解释传入的参数, 之后按照参数执行指定的行为。joptsimple 工具的具体使用这里不做详细说明, 请读者参查阅相关资料。


```

def main(args: Array[String]): Unit = {
    // 解析参数, TopicCommand 支持 list、describe、create、alter、delete 五种操作
    val opts = new TopicCommandOptions(args)
    ..... // 检测参数长度 (略)
    val zkUtils = ZkUtils(opts.options.valueOf(opts.zkConnectOpt),
        30000, 30000, JaasUtils.isZkSecurityEnabled())
    // 根据输入的参数执行不同的操作
    var exitCode = 0
    try {
        if(opts.options.has(opts.createOpt)) // create 参数, 创建 Topic
            createTopic(zkUtils, opts)
        else if(opts.options.has(opts.alterOpt)) // alter 参数, 修改 Topic
            alterTopic(zkUtils, opts)
        else if(opts.options.has(opts.listOpt)) // list 参数, 查询 Topic 名称
            listTopics(zkUtils, opts)
        // describe 参数, 查询 Topic 的详细信息
        else if(opts.options.has(opts.describeOpt))
            describeTopic(zkUtils, opts)
        else if(opts.options.has(opts.deleteOpt)) // delete 参数, 删除 Topic
            deleteTopic(zkUtils, opts)
    } catch {
        ..... // 异常处理 (略)
    } finally {
        zkUtils.close()
        System.exit(exitCode)
    }
}

```

listTopics() 方法和 describeTopic() 方法主要是从 ZooKeeper 中指定的路径查询 Topic 的信息, 逻辑比较简单, 不再做单独的分析。下面来关注创建 Topic 和修改 Topic 的相关逻辑。

5.2.1 创建 Topic

TopicCommand.createTopic() 方法负责创建 Topic, 其核心逻辑是确定新建 Topic 中有多少个分区以及每个分区中的副本如何分配, 这里支持使用 “replica-assignment” 参数手动分配, 也支持使用 “partitions” 参数和 “replication-factor” 参数指定分区个数和副本个

数进行自动分配。之后，该方法会将副本的分配结果写入到 ZooKeeper 中。

```
def createTopic(zkUtils: ZkUtils, opts: TopicCommandOptions) {
  val topic = opts.options.valueOf(opts.topicOpt) // 获取 topic 参数
  // 将 config 参数解析成 Properties 对象
  val configs = parseTopicConfigsToBeAdded(opts)
  // 读取 if-not-exists 参数
  val ifNotExists = if (opts.options.has(opts.ifNotExistsOpt)) true else
false
  // 检测 Topic 名称是否包含 “.” 或 “_” 字符，若包含则输出警告信息（略）
  if (Topic.hasCollisionChars(topic))
    try {
      // 检测是否有 replica-assignment 参数
      if (opts.options.has(opts.replicaAssignmentOpt)) {
        // replica-assignment 参数的格式类似：0:1:2,3:4:5,6:7:8。其中指定了编号为
        // 0 的分区
        // 有三个副本且分配在 Broker0~2 上，编号为1的分区由三个副本且分配在 Broker3~5 上，
        // 后面类似。这里将 replica-assignment 参数内容解析成 Map[Int, Seq[Int]] 格式，
        // 其 key 为分区的编号，value 是其副本所分配的 BrokerId
        val assignment = parseReplicaAssignment(
          opts.options.valueOf(opts.replicaAssignmentOpt))
        ..... // 通过 warnOnMaxMessagesChange() 方法检测 maxMessageBytes 参数
          // 并给出相应提示（略）

        // 对 Topic 名称和副本分配结果进行一系列检测，并写入 ZooKeeper 中
        AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK(zkUtils,
topic,
          assignment, configs, update = false)
      } else {
        // 如果进行副本自动分配，则必须指定 partitions 参数和 replication-factor 参数
        CommandLineUtils.checkRequiredArgs(opts.parser,
          opts.options, opts.partitionsOpt, opts.replicationFactorOpt)
        // 获取 partitions 参数值和 replication-factor 参数值
        val partitions = opts.options.valueOf(opts.partitionsOpt).intValue
        val replicas = opts.options.valueOf(opts.replicationFactorOpt).
intValue
        ..... // 通过 warnOnMaxMessagesChange() 方法检测 maxMessageBytes 参数
          // 并给出相应提示（略）
      }
    }
}
```

```

// 根据 disable-rack-aware 参数决定分配副本时是否考虑机架信息
val rackAwareMode = if (opts.options.has(opts.disableRackAware))
    RackAwareMode.Disabled
else
    RackAwareMode.Enforced

// 自动分配副本，并写入 Zookeeper
AdminUtils.createTopic(zkUtils, topic, partitions,
    replicas, configs, rackAwareMode)
}
} catch {
    // 处理 TopicExistsException 异常 (略)
}
}

```

`AdminUtils.createTopic()` 方法中实现了自动分配副本的功能，首先从 ZooKeeper 中获取 Broker 的信息并封装成 `BrokerMetadata` 集合，`BrokerMetadata` 中只包含 `BrokerId` 和 `rack` (机架) 信息。之后调用 `AdminUtils.assignReplicasToBrokers()` 方法，它会根据上述 Broker 信息和命令参数进行自动分配，其中对于不需要机架感知的分配调用 `AdminUtils.assignReplicasToBrokersRackUnaware()` 方法进行处理，对于需要机架感知的分配调用 `AdminUtils.assignReplicasToBrokersRackAware()` 方法进行处理。最后使用 `AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK()` 方法将分配结果写入到 ZooKeeper 中。

`assignReplicasToBrokersRackUnaware()` 方法的代码如下：

```

private def assignReplicasToBrokersRackUnaware(nPartitions: Int,
    replicationFactor: Int, brokerList: Seq[Int],
    fixedStartIndex: Int, startPartitionId: Int): Map[Int, Seq[Int]] = {
    val ret = mutable.Map[Int, Seq[Int]]() // 用于记录副本分配结果
    val brokerArray = brokerList.toArray
    // 选择起始 Broker 进行分配
    val startIndex = if (fixedStartIndex >= 0) fixedStartIndex
    else rand.nextInt(brokerArray.length)
    var currentPartitionId = math.max(0, startPartitionId) // 选择起始分区

    // nextReplicaShift 指定了副本的间隔，目的是为了更均匀地将副本分配到不同的 Broker 上
    var nextReplicaShift = if (fixedStartIndex >= 0) fixedStartIndex else
        rand.nextInt(brokerArray.length)
}

```



```

for (_ <- 0 until nPartitions) {
  if (currentPartitionId > 0 && (currentPartitionId % brokerArray.length
== 0))
    nextReplicaShift += 1 // 递增 nextReplicaShift
  // 将“优先副本”分配到 startIndex 指定的 Broker 之上
  val firstReplicaIndex = (currentPartitionId + startIndex) % brokerArray.
length
  // 记录“优先副本”的分配结果
  val replicaBuffer = mutable.ArrayBuffer(brokerArray(firstReplicaIndex))
  for (j <- 0 until replicationFactor - 1) // 分配当前分区的其他副本
    replicaBuffer += brokerArray(replicaIndex(firstReplicaIndex,
      nextReplicaShift, j, brokerArray.length))
  ret.put(currentPartitionId, replicaBuffer)
  currentPartitionId += 1 // 分配下一个分区
}
ret
}

```

下面通过一个示例对 assignReplicasToBrokersRackUnaware() 方法的分配过程进行详细说明, 假设现有 5 个 Broker0 ~ 4, 10 个分区 0~9, 每个分区分配三个副本, 无机架信息, 并且 fixedStartIndex 和 startPartitionId 均为 -1, startIndex 随机得到的值为 2, nextReplicaShift 随机得到的值为 1。首先分区 0 的“优先副本”分配到 Broker2 上, 分配其他副本通过 replicaIndex() 方法实现:

```

private def replicaIndex(firstReplicaIndex: Int,
  secondReplicaShift: Int, replicaIndex: Int, nBrokers: Int): Int
= {
  val shift = 1 + (secondReplicaShift + replicaIndex) % (nBrokers - 1)
  (firstReplicaIndex + shift) % nBrokers
}

```

通过 replicaIndex() 方法计算, 分区 0 的第二个副本分配到 Broker4 上, 第三个副本分配到 Broker0 上, 后续其他分区的副本分配与此类似, 其中需要注意的是, Partition0~Partition4 的副本分配分配结束后, 此时 nextReplicaShift 递增变为 2。分区 0 ~ 9 的分配结果如图 5-1 所示:

Broker0	Broker1	Broker2	Broker3	Broker4	
P3	P4	P0	P1	P2	1st
P8	p9	P5	P6	P7	1st
P1	P2	P3	P4	P0	2nd
P5	P6	P7	P8	P9	2nd
P0	P1	P2	P3	P4	3nd
P9	P5	P6	P7	P8	3nd

图 5-1

当需要进行机架感知时, 通过 `assignReplicasToBrokersRackAware()` 方法实现副本分配, 它尽量将每个分区的副本均匀地分配到不同的机架, 如果每个机架上已经有了此分区的副本, 则尽量均匀地分配到每个 Broker 上, 其代码实现如下:

```
private def assignReplicasToBrokersRackAware(nPartitions: Int, replicationFactor:
Int,
    brokerMetadatas: Seq[BrokerMetadata], fixedStartIndex: Int,
    startPartitionId: Int): Map[Int, Seq[Int]] = {
    val brokerRackMap = brokerMetadatas.collect { // 对机架信息进行转换
        case BrokerMetadata(id, Some(rack)) =>
            id -> rack
    }.toMap
    val numRacks = brokerRackMap.values.toSet.size // 统计机架个数
    // 基于机架信息生成一个 Broker 列表, 不同机架上的 Broker 交替出现, 下文通过示例说明
    val arrangedBrokerList = getRackAlternatedBrokerList(brokerRackMap)
    val numBrokers = arrangedBrokerList.size
    val ret = mutable.Map[Int, Seq[Int]]() // 用于记录副本分配结果
    val startIndex = if (fixedStartIndex >= 0) fixedStartIndex
    else
        rand.nextInt(arrangedBrokerList.size) // 选择起始 Broker 进行分配
    var currentPartitionId = math.max(0, startPartitionId) // 选择起始分区
    var nextReplicaShift = if (fixedStartIndex >= 0) fixedStartIndex
    else
        rand.nextInt(arrangedBrokerList.size)
    for (_ <- 0 until nPartitions) {
        if (currentPartitionId > 0 && (currentPartitionId % arrangedBrokerList.
size == 0))
            nextReplicaShift += 1 // 递增 nextReplicaShift
```

```

// 计算 " 优先副本 " 所在的 Broker
val firstReplicaIndex = (currentPartitionId + startIndex) %
arrangedBrokerList.size
val leader = arrangedBrokerList(firstReplicaIndex)
val replicaBuffer = mutable.ArrayBuffer(leader) // 记录 " 优先副本 " 所在的
Broker
// 记录已经分配了当前分区的副本的机架信息
val racksWithReplicas = mutable.Set(brokerRackMap(leader))
// 记录已经分配了当前分区的副本的 Broker 信息
val brokersWithReplicas = mutable.Set(leader)
var k = 0
for (_ <- 0 until replicationFactor - 1) {
    // 分配当前分区的剩余副本
    var done = false
    while (!done) {
        // 通过 replicaIndex() 方法计算当前副本所在的 Broker
        val broker = arrangedBrokerList(replicaIndex(firstReplicaIndex,
            nextReplicaShift * numRacks, k, arrangedBrokerList.size))

        val rack = brokerRackMap(broker)
        // 检测是否跳过此 Broker, 满足下列任一条件即会跳过:
        // 1. 当前机架上已经分配过其他副本, 而且存在机架还未分配副本
        // 2. 当前 Broker 上已经分配过其他副本, 而且存在其他 Broker 还未分配副本
        if ((!racksWithReplicas.contains(rack) ||
            racksWithReplicas.size == numRacks) &&
            (!brokersWithReplicas.contains(broker) ||
            brokersWithReplicas.size == numBrokers)) {
            replicaBuffer += broker // 记录分配结果
            racksWithReplicas += rack // 记录此机架已经分配了当前 Partition 的副本
            // 记录此 Broker 已经分配了当前 Partition 的副本
            brokersWithReplicas += broker
            done = true // 标识完成此副本的分配
        }
        k += 1
    }
}
ret.put(currentPartitionId, replicaBuffer)
currentPartitionId += 1

```

```

}
ret
}

```

这里依然按照前面对 `assignReplicasToBrokersRackUnaware()` 方法分析时使用的示例进行说明, 这里将 Broker0~1 划分到 rack1 上, 将 Broker2~3 划分到 rack2 上, 将 Broker4 划分到 rack3 上, 得到 `arrangedBrokerList` 列表的值为 [0, 2, 4, 1, 3], 很明显它是轮训每个机架上的 Broker 产生的列表。

按照上述条件分区 0 和分区 1 的副本分配结果分别为 [Broker4, Broker2, Broker1] 和 [Broker1, Broker4, Broker3]。在分配分区 2 的分配过程中会出现跳过 Broker 的情况, 分区 2 的“优先副本”分配到 Broker3, 第二个副本分配到 Broker1, 在第三个副本时首先会尝试分配到 Broker0, 因与第二个副本分配在同一机架且 rack3 上并没有分配分区 2 的副本, 所以需重新尝试 Broker2, 依然失败, 直到尝试到 Broker4 时才成功分配。由于 rack3 中只有一个 Broker, 所以 Broker4 上会有所有分区的副本。最终的分配结果如图 5-2 所示。

rack1		rack2		rack3	
Broker0	Broker1	Broker2	Broker3	Broker4	
P3	P1	P4	P2	P0	1st
P8	P6	P9	P7	P5	1st
P4	P2	P0	P3	P1	2nd
P5		P6	P8	P7	2nd
P9					2nd
	P0	P5	P1	P2	3nd
	P7			P3	3nd
				P4	3nd
				P6	3nd
				P8	3nd
				P9	3nd

图 5-2

无论是通过“`replica-assignment`”参数手动分配副本还是通过上述方式自动分配副本, 最终都会调用 `AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK()` 方法将副本的分配结果写入 ZooKeeper 中保存, 其代码如下:


```

def createOrUpdateTopicPartitionAssignmentPathInZK(zkUtils: ZkUtils,
topic: String,
    partitionReplicaAssignment: Map[Int, Seq[Int]],
    config: Properties = new Properties, update: Boolean = false) {
    ..... // 检测 Topic 名称 (略), 检测所有分区的副本数是一致的 (略)
    // 获取当前 Topic 在 ZooKeeper 中对应的存储路径, 即 "/brokers/topics/[topic_name]"
    val topicPath = getTopicPath(topic)
    ..... // 对于新创建的 Topic, 需要检测 Zookeeper 中是否已存在了同名的 Topic (略)

    // 检测是否存在对是否出现同一个 Broker 上分配多个副本的情况
    partitionReplicaAssignment.values.foreach(
        reps => require(reps.size == reps.toSet.size,
            "Duplicate replica assignment found: " + partitionReplicaAssignment))

    if (!update) { // 在创建 Topic 时, 会将 config 写入 ZooKeeper
        writeEntityConfig(zkUtils, ConfigType.Topic, topic, config)
    }
    // 将副本的分配转换成 JSON 格式写入 ZooKeeper
    writeTopicPartitionAssignment(zkUtils, topic, partitionReplicaAssignment,
update)
}

```

需要读者了解的是, 除了使用 kafka-topics 脚本, 在 KafkaApis 中需要创建 Topic 时, 例如开启了自动创建 Topic 的配置, 也会调用 AdminUtils.createTopic() 方法, 如图 5-3 所示。

```

▼ ④ AdminUtils$.createTopic(ZkUtils, String, int, int, Properties, RackAwareMode) (kafka.admin)
  ► ④ TopicCommand$.createTopic(ZkUtils, TopicCommandOptions) (kafka.admin)
  ▼ ④ KafkaApis.createTopic(String, int, int, Properties) (kafka.server)
    ▼ ④ KafkaApis.getTopicMetadata(Set<String>, SecurityProtocol, boolean) (kafka.server)
      ► ④ KafkaApis.handleTopicMetadataRequest(Request) (kafka.server)
    ▼ ④ KafkaApis.createGroupMetadataTopic() (kafka.server)
      ▼ ④ KafkaApis.getOrCreateGroupMetadataTopic(SecurityProtocol) (kafka.server)
        ► ④ KafkaApis.handleGroupCoordinatorRequest(Request) (kafka.server)
      ▼ ④ KafkaApis.getTopicMetadata(Set<String>, SecurityProtocol, boolean) (kafka.server)
        ► ④ KafkaApis.handleTopicMetadataRequest(Request) (kafka.server)

```

图 5-3

5.2.2 修改 Topic

TopicCommand.alterTopic() 方法负责修改 Topic 的分区数量、副本的分配以及相关配

置信息。需要注意，对于修改 Topic 配置项的逻辑已经转移到 kafka-configs 脚本中了，不再推荐使用 kafka-topics 脚本完成该功能。alterTopic() 方法的具体实现如下：

```
def alterTopic(zkUtils: ZkUtils, opts: TopicCommandOptions) {
  // 从 ZooKeeper 中获取与“topic”参数正则匹配的 Topic 集合，这里并不会抛出内部 Topic
  val topics = getTopics(zkUtils, opts)
  // 读取 if-exists 配置
  val ifExists = if (opts.options.has(opts.ifExistsOpt)) true else false
  if (topics.length == 0 && !ifExists) {
    ... .. // 抛出异常（略）
  }
  topics.foreach { topic =>
    ... ..// 修改 Topic 配置项信息的功能（略）
    if (opts.options.has(opts.partitionsOpt)) {
      // 检测是否包含“partitions”参数
      ... .. // 若要修改 Offsets Topic 这个内部 Topic 的分区数量，抛出异常（略）
      val nPartitions = opts.options.valueOf(opts.partitionsOpt).intValue
      // 获取“replica-assignment”参数的值
      val replicaAssignmentStr = opts.options.valueOf(opts.replicaAssignmentOpt)
      // 调用 AdminUtils.addPartitions() 方法完成分区数量的增加以及副本分配
      AdminUtils.addPartitions(zkUtils, topic, nPartitions, replicaAssignmentStr)
    }
  }
}
```

AdminUtils.addPartitions() 方法会根据原有分区的分配情况确定副本个数，根据是否指定 replica-assignment 参数决定新增分区是否进行自动副本分配，最后将原有分区和新增的 Partition 的副本分配结果合并后写入 ZooKeeper。

```
def addPartitions(zkUtils: ZkUtils, topic: String, numPartitions: Int = 1,
  replicaAssignmentStr: String = "", checkBrokerAvailable:
  Boolean = true,
  rackAwareMode: RackAwareMode = RackAwareMode.Enforced) {
  // 从 ZooKeeper 获取此 Topic 当前的副本分配情况
  val existingPartitionsReplicaList =
    zkUtils.getReplicaAssignmentForTopics(List(topic))
  if (existingPartitionsReplicaList.size == 0)
```

```

    throw new AdminOperationException("The topic %s does not exist".
format(topic))

// 获取分区 0 的副本分配情况
val existingReplicaListForPartitionZero = existingPartitionsReplicaList.
find(p =>
    p._1.partition == 0) match {
    case None => ... .. // 抛出异常(略)
    case Some(headPartitionReplica) => headPartitionReplica._2
}
// 获取分区 0 的副本数量
val partitionsToAdd = numPartitions - existingPartitionsReplicaList.size

..... // 只能增加分区数量, 如果指定的分区数量小于当前分区数, 则抛出异常(略)
// create the new partition replication list
val brokerMetadatas = getBrokerMetadatas(zkUtils, rackAwareMode)
val newPartitionReplicaList =
    if (replicaAssignmentStr == null || replicaAssignmentStr == "") {
        // 确定 startIndex
        val startIndex = math.max(0, brokerMetadatas.indexWhere(_.id >=
            existingReplicaListForPartitionZero.head))
        // 对新增分区进行副本自动分配, 注意 fixedStartIndex 和 startPartitionId 参数的取值
        // AdminUtils.assignReplicasToBrokers() 方法前面已经分析过了, 这里不再赘述
        AdminUtils.assignReplicasToBrokers(brokerMetadatas, partitionsToAdd,
            existingReplicaListForPartitionZero.size, startIndex,
            existingPartitionsReplicaList.size)
    } else
        // 解析 replica-assignment 参数, 其中会进行一系列有效性检测
        getManualReplicaAssignment(replicaAssignmentStr,
            brokerMetadatas.map(_.id).toSet, existingPartitionsReplicaList.
size,
            checkBrokerAvailable)

..... // 检测新增 Partition 的副本数是否正常(略)
// 将原有分区与新增分区的副本分配整理成集合
val partitionReplicaList = existingPartitionsReplicaList.map(p =>
    p._1.partition -> p._2)

```

```

partitionReplicaList += newPartitionReplicaList
// 将最终的副本分配结果写入 ZooKeeper, 注意最后一个参数的值, 这里只更新副本分配情况
AdminUtils.createOrUpdateTopicPartitionAssignmentPathInZK(zkUtils, topic,
    partitionReplicaList, update = true)
}

```

在第 4 章介绍过, 删除 Topic 由 TopicDeletionManager 负责。TopicCommand.deleteTopic() 方法主要负责将待删除的 Topic 名称写入到 Zookeeper 的 “/admin/delete_topics” 路径下, 这会触发 DeleteTopicsListener 将待删除 Topic 交由 TopicDeletionManager 处理。deleteTopic() 方法比较简单, 这里就不再贴出代码了。

5.3 kafka-preferred-replica-election 脚本

在介绍 KafkaController 时分析了 PreferredReplicaElectionListener 的实现, 它会监听 ZooKeeper 中的 “/admin/preferred_replica_election” 节点, 负责对指定的分区进行“优先副本”选举。而这里“指定的分区”可以通过 kafka-preferred-replica-election 脚本写入到 ZooKeeper, 该脚本是通过调用 PreferredReplicaLeaderElectionCommand 实现的, 其 path-to-json-file 参数指定一个 JSON 格式的输入文件, 在其中指定了需要进行“优先副本”选举的分区, 该 JSON 格式的输入文件示例如下:

```

{
  "partitions": [
    {
      "topic": "foo", "partition": 1
    },
    {
      "topic": "foobar", "partition": 2
    }
  ]
}

```

如果未指定输入文件, 则认为所有分区都需要进行“优先副本”选举操作。

首先来分析 PreferredReplicaLeaderElectionCommand.main() 方法, 它首先会检测 path-to-json-file 参数, 决定需要进行“优先副本”选举的分区集合, 之后会将该分区集合交给 PreferredReplicaLeaderElectionCommand 处理。


```

def main(args: Array[String]): Unit = {
    ... ..// 检测 path-to-json-file 参数和 zookeeper 参数 (略)
    try {
        // 创建与 ZooKeeper 的连接
        zkClient = ZkUtils.createZkClient(zkConnect, 30000, 30000)
        zkUtils = ZkUtils(zkConnect, 30000, 30000, JaasUtils.isZkSecurityEnabled())
        // 获取需要进行“优先副本”选举的分区集合
        val partitionsForPreferredReplicaElection =
            // 未指定 path-to-json-file 参数则返回全部分区
            if (!options.has(jsonFileOpt))
                zkUtils.getAllPartitions()
            else
                // 解析 path-to-json-file 参数指定的输入文件
                parsePreferredReplicaElectionData(Utils.readFileAsString(
                    options.valueOf(jsonFileOpt)))
        // 创建 preferredReplicaElectionCommand 对象
        val preferredReplicaElectionCommand = new PreferredReplicaLeaderElecti
onCommand(
            zkUtils, partitionsForPreferredReplicaElection)

        // 将指定的分区写入到 ZooKeeper 中的 “/admin/preferred_replica_election” 节点中
        preferredReplicaElectionCommand.moveLeaderToPreferredReplica()
    } catch {
        // 异常处理 (略)
    } finally {
        // 关闭与 ZooKeeper 的连接 (略)
    }
}

```

`moveLeaderToPreferredReplica()` 方法首先会检测指定的 Topic 是否包含指定的分区, 之后会调用 `writePreferredReplicaElectionData()` 方法将需要进行“优先副本”选举的分区信息写入 ZooKeeper。


```

def writePreferredReplicaElectionData(zkUtils: ZkUtils,
                                     partitionsUndergoingPreferredReplica
Election: Set[TopicAndPartition]) {
  // “/admin/preferred_replica_election” 节点的路径
  val zkPath = ZkUtils.PreferredReplicaLeaderElectionPath
  val partitionsList = partitionsUndergoingPreferredReplicaElection
    .map(e => Map("topic" -> e.topic, "partition" -> e.partition))
  // 转换成 JSON 格式
  val jsonData = Json.encode(Map("version" -> 1, "partitions" ->
partitionsList))
  try {
    zkUtils.createPersistentPath(zkPath, jsonData) // 写入 Zookeeper
  } catch {
    ..... // 异常处理 (略)
  }
}

```

5.4 kafka-reassign-partitions 脚本

在介绍 `KafkaController` 时分析过 `PartitionsReassignedListener` 的实现，它会监听 ZooKeeper 中的 “/admin/reassign_partitions” 节点，负责进行分区中副本的重新分配。

`kafka-reassign-partitions` 脚本的主要功能有三个：一是生成副本迁移的方案；二是将副本迁移方案写入到 ZooKeeper 中，由 `PartitionsReassignedListener` 处理；三是检测指定分区的副本迁移是否完成。

`ReassignPartitionsCommand.main()` 方法是 `kafka-reassign-partitions` 脚本的入口方法，它会检测 `verify`、`generate`、`execute` 三个参数并进入不同的处理流程。

```

def main(args: Array[String]): Unit = {
  val opts = new ReassignPartitionsCommandOptions(args)
  ..... // 检测 generate、execute、verify 三个参数是否只出现一个，若出现两个以上，
    // 则异常结束 (略)
  // 创建与 ZooKeeper 的连接
  val zkConnect = opts.options.valueOf(opts.zkConnectOpt)
  val zkUtils = ZkUtils(zkConnect, 30000, 30000, JaasUtils.isZkSecurityEnabled())
  try {

```

```

if(opts.options.has(opts.verifyOpt))
    verifyAssignment(zkUtils, opts) // 检测副本迁移是否结束
else if(opts.options.has(opts.generateOpt))
    generateAssignment(zkUtils, opts) // 输出副本迁移方案和当前副本的分配情况
else if (opts.options.has(opts.executeOpt))
    executeAssignment(zkUtils, opts) // 执行指定的副本迁移
} catch {
    // 异常处理 (略)
} finally {
    // 关闭与 Zookeeper 的连接
}
}

```

ReassignPartitionsCommand.generateAssignment() 方法负责输出副本迁移方案以及当前分区的 AR 信息。它首先检测 topics-to-move-json-file 参数和 broker-list 参数是否都存在, 不存在则异常结束; 之后解析 topics-to-move-json-file 参数和 broker-list 参数, 读取 topics-to-move-json-file 参数指定的输入文件, 读取 disable-rack-aware 参数; 最后调用 generateAssignment() 方法的重载获取副本迁移的目标以及当前分配的情况。generateAssignment() 的代码如下:

```

def generateAssignment(zkUtils: ZkUtils, brokerListToReassign: Seq[Int],
    topicsToMoveJsonString: String, disableRackAware:
Boolean):
(Map[TopicAndPartition, Seq[Int]], Map[TopicAndPartition, Seq[Int]]) = {
    // 获取待迁移的 Topic
    val topicsToReassign = zkUtils.parseTopicsData(topicsToMoveJsonString)
    // 检测待迁移的 Topic 集合中是否存在重复的 Topic, 若存在则抛出异常 (略)
    // 获取待迁移 Topic 的当前副本分配情况
    val currentAssignment = zkUtils.getReplicaAssignmentForTopics(topicsToReassign)
    // 按照 Topic 名称进行分组
    val groupedByTopic = currentAssignment.groupBy { case (tp, _) =>
tp.topic }
    // 决定是否考虑机架信息
    val rackAwareMode = if (disableRackAware)
        RackAwareMode.Disabled
    else
        RackAwareMode.Enforced
}

```

```

// 获取 broker-list 参数指定的 Broker 信息（其中包括机架信息）
val brokerMetadatas = AdminUtils.getBrokerMetadatas(zkUtils,
    rackAwareMode, Some(brokerListToReassign))

// 记录副本迁移的目标
val partitionsToBeReassigned = mutable.Map[TopicAndPartition, Seq[Int]]()
groupedByTopic.foreach { case (topic, assignment) => // 遍历每个待迁移的
Topic
    val (_, replicas) = assignment.head
    // 进行副本自动分配, assignReplicasToBrokers() 方法已经在前面分析过了, 这里注意,
    // brokerMetadatas 集合中只有 broker-list 参数指定的 Broker 信息
    val assignedReplicas = AdminUtils.assignReplicasToBrokers(brokerMetadatas,
        assignment.size, replicas.size)
    // 记录此 Topic 副本迁移后的结果
    partitionsToBeReassigned += assignedReplicas.map { case (partition,
replicas) =>
        (TopicAndPartition(topic, partition) -> replicas)
    }
}
// 返回生成的副本迁移目标和当前副本分配情况
(partitionsToBeReassigned, currentAssignment)
}

```

ReassignPartitionsCommand.executeAssignment() 方法首先检测 reassignment-json-file 参数是否都存在, 若不存在则异常结束; 之后解析 reassignment-json-file 参数并读取其指定的输入文件, 得到副本迁移方案; 最后调用 executeAssignment() 方法的重载实现将副本迁移方案写入 ZooKeeper。executeAssignment() 方法的具体实现如下:

```

def executeAssignment(zkUtils: ZkUtils, reassignmentJsonString: String) {
    // 解析 reassignment-json-file 参数指定的输入文件内容, 得到待迁移的 Topic 和分区信息
    val partitionsToBeReassigned = zkUtils.parsePartitionReassignmentDataWithoutDedup(
        reassignmentJsonString)
    if (partitionsToBeReassigned.isEmpty) // 检测输入文件内容是否为空
        throw new AdminCommandFailedException(...)
    val duplicateReassignedPartitions = CoreUtils.duplicates(partitionsToBeReassigned
        .map { case (tp, replicas) => tp })
    if (duplicateReassignedPartitions.nonEmpty) // 检测待迁移 Topic 是否存在重复

```



```

    throw new AdminCommandFailedException("...")
    val duplicateEntries = partitionsToBeReassigned
      .map { case (tp, replicas) => (tp, CoreUtils.duplicates(replicas)) }
      .filter { case (tp, duplicatedReplicas) => duplicatedReplicas.nonEmpty }
    if (duplicateEntries.nonEmpty) { // 检测待迁移分区的副本分配是否存在重复
      val duplicatesMsg = duplicateEntries
        .map { case (tp, duplicateReplicas) => "%s contains multiple entries
for %s"
          .format(tp, duplicateReplicas.mkString(", "))
        }.mkString(". ")
      throw new AdminCommandFailedException("...")
    }

    // 创建 ReassignPartitionsCommand
    val reassignPartitionsCommand = new ReassignPartitionsCommand(zkUtils,
      partitionsToBeReassigned.toMap)
    ... .. // 在开始迁移之前, 输出当前分区的副本分配情况 (略)
    // 调用 reassignPartitions() 方法, 将迁移的目标写入到
    // “/admin/reassign_partitions” 路径
    if (reassignPartitionsCommand.reassignPartitions())
      println("Successfully started reassignment of partitions...")
    else
      println("Failed to reassign partitions...")
  }
}

```

当迁移的目标信息写入到“/admin/reassign_partitions”路径后, 会触发 PartitionsReassignedListener 监听器, 其相关处理请读者参考之前的分析。

ReassignPartitionsCommand.verifyAssignment() 主要负责检测副本迁移操作是否已完成。它首先读取并分析 reassignment-json-file 参数指定的输入文件内容, 之后调用 checkIfReassignmentSucceeded() 方法检测每个分区的迁移情况。

```

private def checkIfReassignmentSucceeded(zkUtils: ZkUtils,
  partitionsToBeReassigned: Map[TopicAndPartition, Seq[Int]]):
  Map[TopicAndPartition, ReassignmentStatus] = {
  // 从 “/admin/reassign_partitions” 路径下获取正在执行副本重新分配的分区信息
  val partitionsBeingReassigned =
    zkUtils.getPartitionsBeingReassigned().mapValues(_._newReplicas)
}

```



```

// 对每个分区都调用 checkIfPartitionReassignmentSucceeded() 方法检测迁移是否成功
partitionsToBeReassigned.map { topicAndPartition =>
    (topicAndPartition._1,
     checkIfPartitionReassignmentSucceeded(zkUtils, topicAndPartition._1,
                                             topicAndPartition._2, partitionsToBeReassigned, partitionsBeingReassigned))
}
}

// 下面是 checkIfPartitionReassignmentSucceeded() 方法的实现
def checkIfPartitionReassignmentSucceeded(zkUtils: ZkUtils,
    topicAndPartition: TopicAndPartition, reassignedReplicas: Seq[Int],
    partitionsToBeReassigned: Map[TopicAndPartition, Seq[Int]],
    partitionsBeingReassigned: Map[TopicAndPartition, Seq[Int]]):
    ReassignmentStatus = {
// 获取分区中副本重新分配的目标, 若已经分配完毕, 则返回空
val newReplicas = partitionsToBeReassigned(topicAndPartition)
partitionsBeingReassigned.get(topicAndPartition) match {
    // ZooKeeper 中的 "/admin/reassign_partitions" 路径下依然有此分区信息, 则表示
    // 正在进行副本重新分配
    case Some(partition) => ReassignmentInProgress
    case None =>
        // 当前 AR 与迁移目标一致, 即为重新分配成功
        val assignedReplicas = zkUtils.getReplicasForPartition(topicAndPartition.
topic,
    topicAndPartition.partition)
        if (assignedReplicas == newReplicas) ReassignmentCompleted
        else {
            // 迁移已经完成, 但当前 AR 与目标不一致, 则表示迁移失败
            ReassignmentFailed
        }
    }
}
}

```

灵活使用 `kafka-reassign-partitions` 脚本可以完成很多常用的功能, 例如, 将待下线 Broker 中的全部副本迁移到集群中其他 Broker 上, 即可实现 Broker 平滑下线的功能; 当需要提高某些分区的容灾能力时, 可以在进行副本迁移时修改副本数量。

5.5 kafka-console-producer 脚本

kafka-console-producer 脚本是一个简易的控制台版本的生产者，可以通过在控制台中输入消息的 key 和 value，然后由此脚本生成请求将消息追加到 Kafka 服务端。kafka-console-producer 脚本的功能通过调用 ConsoleProducer 实现，其中同时支持新旧两个版本的生产者，这里主要分析新版本生产者的相关实现。

ConsoleProducer.main() 方法是 kafka-console-producer 脚本的入口函数，代码如下：

```
def main(args: Array[String]) {
  try {
    val config = new ProducerConfig(args) // 读取命令行参数并进行解析
    // 创建 LineMessageReader 对象
    val reader = Class.forName(config.readerClass).newInstance()
      .asInstanceOf[MessageReader]
    // 初始化 LineMessageReader 对象
    reader.init(System.in, getReaderProps(config))
    val producer =
      if (config.useOldProducer) {
        new OldProducer(getOldProducerProps(config))
      } else {
        // 重点分析新版本的 Producer
        new NewShinyProducer(getNewProducerProps(config))
      }

    Runtime.getRuntime.addShutdownHook(new Thread() { // 添加 JVM 的关闭钩子
      override def run() {
        producer.close()
      }
    })

    var message: ProducerRecord[Array[Byte], Array[Byte]] = null
    do {
      message = reader.readMessage() // 读取控制台数据，形成 ProducerRecord
      if (message != null)
        producer.send(message.topic, message.key, message.value) // 发送消息
    } while (message != null)
  } catch {
    ..... // 异常处理（略）
  }
}
```

ConsoleProducer 使用 LineMessageReader 读取控制台的输入，其中封装了在 init() 方法中传入的 System.in 输入流，LineMessageReader.readMessage() 可以根据配置决定是否以及如何切分控制台输入的 key 和 value，默认按照 “\t” 进行切分，并返回 ProducerRecord。代码比较简单，就不贴出来了。

NewShinyProducer 中封装了第 2 章中介绍的 KafkaProducer 对象，并依赖 KafkaProducer 完成发送消息的功能。

```
class NewShinyProducer(producerProps: Properties) extends BaseProducer {
    val sync = producerProps.getProperty("producer.type", "async").
equals("sync")
    val producer = new KafkaProducer[Array[Byte],Array[Byte]](producerProps)

    override def send(topic: String, key: Array[Byte], value: Array[Byte]) {
        val record = new ProducerRecord[Array[Byte],Array[Byte]](topic, key,
value)
        if(sync) { // 同步发送
            this.producer.send(record).get()
        } else { // 异步发送
            this.producer.send(record, new ErrorLoggingCallback(topic, key,
value, false))
        }
    }
}
```

5.6 kafka-console-consumer 脚本

kafka-console-consumer 脚本是一个简易的控制台 Consumer。此脚本的功能通过调用 ConsoleConsumer 实现，其中同时支持新旧两个版本的消费者，这里主要分析新版本 Consumer 的相关实现。

ConsoleConsumer.run() 方法是 kafka-console-consumer 脚本核心方法，实现如下：

```
def run(conf: ConsumerConfig) {
    val consumer = if (conf.useNewConsumer) {
        val timeoutMs = if (conf.timeoutMs >= 0) conf.timeoutMs else Long.
MaxValue
        // NewShinyConsumer 中封装了 KafkaConsumer 对象
```



```

    new NewShinyConsumer(Option(conf.topicArg), Option(conf.whitelistArg),
        getNewConsumerProps(conf), timeoutMs)
  } else {
    ..... // 旧版本 Consumer (略)
  }
  // 添加 JVM 关闭钩子, 用于关闭 NewShinyConsumer
  addShutdownHook(consumer, conf)
  try {
    // 从服务端获取消息并输出
    process(conf.maxMessages, conf.formatter, consumer, conf.skipMessageOnError)
  } finally {
    ..... // 关闭 KafkaConsumer 以及清理 ZooKeeper 的相关操作 (略)
  }
}

```

在 NewShinyConsumer 中封装了 KafkaConsumer 对象, 并依赖 KafkaConsumer 实现从服务端拉去消息的功能, 具体实现如下:

```

class NewShinyConsumer(topic: Option[String], whitelist: Option[String],
    consumerProps: Properties, val timeoutMs: Long = Long.MaxValue)
    extends BaseConsumer {
  // 创建 KafkaConsumer
  val consumer = new KafkaConsumer(Array[Byte], Array[Byte])(consumerProps)
  if (topic.isDefined) // 订阅指定的 Topic
    consumer.subscribe(List(topic.get))
  else if (whitelist.isDefined) // 未指定 Topic 则订阅白名单中指定的 Topic
    consumer.subscribe(Pattern.compile(whitelist.get),
        new NoOpConsumerRebalanceListener())
  else // 未指定 Topic 和白名单, 则抛出异常
    throw new IllegalArgumentException("...")

  var recordIter = consumer.poll(0).iterator // 初始化时, 尝试从服务端获取消息

  override def receive(): BaseConsumerRecord = {
    if (!recordIter.hasNext) {
      // 上次拉取的消息全部处理后, 则继续从服务端拉取消息
      recordIter = consumer.poll(timeoutMs).iterator
      if (!recordIter.hasNext)
        throw new ConsumerTimeoutException
    }
  }
}

```



```

    }
    val record = recordIter.next
    // 将消息封装成 BaseConsumerRecord 返回
    BaseConsumerRecord(record.topic, record.partition, record.offset,
        record.timestamp, record.timestampType, record.key, record.value)
    }
    ... ..
}

```

ConsoleConsumer.process() 方法中会调用 NewShinyConsumer.receive() 方法获取消息，之后调用 MessageFormatter.writeTo() 方法将消息按照指定格式输出。MessageFormatter 有多个子类，可用于处理不同类型的消息以及实现不同的输出内容，如图 5-4 所示。MessageFormatter 各个子类的功能如下所述。

- DefaultMessageFormatter: 输出消息的 key 和 value。
- LoggingMessageFormatter: 封装了 DefaultMessageFormatter，除了输出消息的 key 和 value，还会输出日志信息。
- NoOpMessageFormatter: 空实现，不会输出任何内容。
- ChecksumMessageFormatter: 输出消息的校验码。
- OffsetsMessageFormatter: 主要在消费 Offsets Topic 这个内部 Topic 时使用，它会解析并输出记录 Offset 相关信息的信息。
- GroupMetadataMessageFormatter: 主要在消费 Offsets Topic 这个内部 Topic 时使用，它会解析并输出记录 Consumer Group 信息的信息。

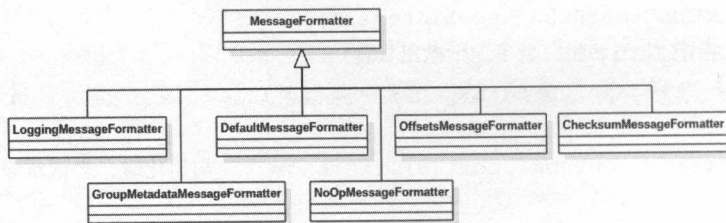


图 5-4

默认使用的是 DefaultMessageFormatter 实现。MessageFormatter 这些子类的实现并不复杂，请读者参考源码学习。

5.7 kafka-consumer-groups 脚本

kafka-consumer-groups 脚本的主要功能有三个：一是查询当前所有 Consumer Group；二是获取某 Consumer Group 的详细信息；三是删除某 Consumer Group。注意，旧版本 Consumer 中的 Consumer Group 相关信息保存在 ZooKeeper 中，该脚本只能删除旧版本的 Consumer Group；新版本消费者的 Consumer Group 信息记录在 Offsets Topic 这个内部 Topic 中，不能通过该脚本进行删除。ConsumerGroupCommand.main() 方法是 kafka-consumer-groups 脚本的入口函数，具体实现如下：

```
def main(args: Array[String]) {
    .....// 检测参数个数,list参数、describe参数、delete参数同时出现多个则抛出异常(略)
    val consumerGroupService = {
        // 通过 new-consumer 参数指定使用新版本的还是旧版本的消费者
        if (opts.options.has(opts.newConsumerOpt))
            // 新版本消费者对应的 ConsumerGroupService 实现
            new KafkaConsumerGroupService(opts)
        else
            // 旧版本消费者对应的 ConsumerGroupService 实现
            new ZkConsumerGroupService(opts)
    }

    try {
        if (opts.options.has(opts.listOpt))
            consumerGroupService.list() // 输出全部 Consumer Group 的 id
        else if (opts.options.has(opts.describeOpt))
            consumerGroupService.describe() // 获取指定 Consumer Group 的描述信息
        else if (opts.options.has(opts.deleteOpt)) { // 删除指定的 Consumer Group
            consumerGroupService match {
                case service: ZkConsumerGroupService => service.delete()
            }
            // 新版本消费者对应 KafkaConsumerGroupService, 不支持删除 Consumer Group 的操作
            case _ => throw new IllegalStateException("...")
        }
    } catch {
        ..... // 异常处理(略)
    } finally {
        consumerGroupService.close()
    }
}
```

KafkaConsumerGroupService 是新版本消费者对应的 ConsumerGroupService 实现，这里重点分析此实现，ZkConsumerGroupService 是旧版本消费者对应的 ConsumerGroupService 实现，这里不做详细分析。

KafkaConsumerGroupService 有两个比较重要的字段。

- adminClient: AdminClient 类型，list() 方法和 describe() 方法中都是通过调用 AdminClient 的对应方法实现的。
- consumer: KafkaConsumer 对象，由于只在 describe() 方法中会用到，会延迟创建。延迟创建的代码如下所示。

```
private var consumer: KafkaConsumer[String, String] = null
private def getConsumer() = {
  if (consumer == null)
    consumer = createNewConsumer()
  consumer
}
```

AdminClient 对象是通过其 create() 方法创建的，其中封装了 ConsumerNetworkClient 对象，ConsumerNetworkClient 实现及其底层原理在第 3 章中已经介绍了。

```
def create(config: AdminConfig): AdminClient = {
  ..... // 创建构造 KSelector 和 NetworkClient 需要的对象，例如：Metrics 对象、
         // ChannelBuilder 对象等（略）
  val selector = new Selector(...)
  val networkClient = new NetworkClient(...)
  // 创建 ConsumerNetworkClient
  val highLevelClient = new ConsumerNetworkClient(networkClient, metadata,
    time, DefaultRetryBackoffMs, DefaultRequestTimeoutMs)

  // 创建 AdminClient
  new AdminClient(time, DefaultRequestTimeoutMs, highLevelClient,
    bootstrapCluster.nodes().asScala.toList)
}
```

AdminClient.send() 方法在 ConsumerNetworkClient.send() 方法之上实现了同步发送请求的功能。AdminClient.sendAnyNode() 方法在 send() 方法之上做了一层封装，它会尝试向集群中已知的所有 Broker 发送请求，任一 Broker 成功响应后则返回。


```

private def send(target: Node, api: ApiKeys, request: AbstractRequest): Struct = {
  var future: RequestFuture[ClientResponse] = null
  // 调用 ConsumerNetworkClient.send() 方法发送请求
  future = client.send(target, api, request)
  client.poll(future) // 阻塞等待响应
  if (future.succeeded()) // 根据响应是否异常, 进行相应处理
    return future.value().responseBody()
  else
    throw future.exception()
}

private def sendAnyNode(api: ApiKeys, request: AbstractRequest): Struct = {
  bootstrapBrokers.foreach { case broker => // 遍历集群中已知的 Broker
    try {
      return send(broker, api, request)
    } catch {
      ... ..// 输出日志(略)
    }
  }
  throw new RuntimeException("...")
}

```

KafkaConsumerGroupService.list() 方法是通过调用 AdminClient.listAllGroups() 方法实现的。它首先调用 findAllBrokers() 方法（底层通过 sendAnyNode() 实现）向集群中任一已知的 Broker 发送 MetadataRequest 请求，得到集群中的 Broker 信息。然后尝试向集群中每个 Broker 发送 ListGroupsRequest 请求，获取其负责管理的 ConsumerGroup。

```

def listAllGroups(): Map[Node, List[GroupOverview]] = {
  // 发送 MetadataRequest 请求获取集群中所有 Broker 的信息
  findAllBrokers.map {
    case broker => broker -> {
      try {
        listGroups(broker) // 发送 ListGroupsRequest 并阻塞等待响应
      } catch {
        ..... // 输出日志操作(略)
        List[GroupOverview]()
      }
    }
  }
  }.toMap
}

```


`KafkaConsumerGroupService.describeGroup()` 方法负责获取指定 Consumer Group 的详细信息，其关键步骤如下：

(1) 首先调用 `AdminClient.findCoordinator()` 方法查找指定 Consumer Group 对应的 `GroupCoordinator` 所在的节点。

(2) 向该 `GroupCoordinator` 发送 `DescribeGroupsRequest` 请求查询指定 `ConsumerGroup` 对应的 `GroupMetadata` 信息，并进行解析。

```
def describeGroup(groupId: String): GroupSummary = {
  // 发送 GroupCoordinatorRequest 请求查找 GroupCoordinator
  val coordinator = findCoordinator(groupId)
  // 发送 DescribeGroupsRequest 请求
  val responseBody = send(coordinator, ApiKeys.DESCRIBE_GROUPS,
    new DescribeGroupsRequest(List(groupId).asJava))
  val response = new DescribeGroupsResponse(responseBody)
  val metadata = response.groups().get(groupId)
  .....// 检查 metadata 是否为空，检查响应的错误码（略）

  // 解析 GroupMetadata 对象，并封装成 GroupSummary
  val members = metadata.members().map { member =>
    val metadata = Utils.readBytes(member.memberMetadata())
    val assignment = Utils.readBytes(member.memberAssignment())
    MemberSummary(member.memberId(), member.clientId(),
      member.clientHost(), metadata, assignment)
  }.toList
  GroupSummary(metadata.state(), metadata.protocolType(),
    metadata.protocol(), members)
}
```

(3) 对于 `Stable` 状态的 `ConsumerGroup`，将对应的 `GroupMetadata` 中记录的 `Member` 信息转换成 `ConsumerSummary` 并返回。对于处于其他状态的 `ConsumerGroup`，则返回空集合。

(4) 通过 `KafkaConsumer.committed()` 方法获取指定分区的 `committed offset`，底层通过发送 `OffsetFetchRequest` 请求实现。

(5) 通过 `KafkaConsumer.seekToEnd()` 方法和 `position()` 获取对应分区的 `LEO`。

(6) 最后调用 `describeTopicPartition()` 方法完成输出。

```

protected def describeGroup(group: String) {
  val consumerSummaries = adminClient.describeConsumerGroup(group)
  if (consumerSummaries.isEmpty) println("....")
  else {
    val consumer = getConsumer() // 获取 KafkaConsumer
    consumerSummaries.foreach { consumerSummary =>
      val topicPartitions = consumerSummary.assignment.map(tp =>
        TopicAndPartition(tp.topic, tp.partition))
      // 调用 KafkaConsumer.committed() 方法获取指定分区最近一次提交的 offset, 底层
      // 通过发送 OffsetFetchRequest 请求实现
      val partitionOffsets = topicPartitions.flatMap { topicPartition =>
        Option(consumer.committed(new TopicPartition(topicPartition.topic,
          topicPartition.partition)))
      }.map { offsetAndMetadata =>
        topicPartition -> offsetAndMetadata.offset
      }
      }.toMap
      // 获取分区对应的 LEO 值, 并将所有信息输出
      describeTopicPartition(group, topicPartitions, partitionOffsets.get,
        _ => Some(s"${consumerSummary.clientId}_${consumerSummary.
clientHost}"))
    }
  }
}

```

kafka-consumer-offset-checker 脚本从 Kafka 0.9.0 版本开始已经被废弃了, 其功能已经被本节介绍的 kafka-consumer-groups 脚本所取代。

5.8 DumpLogSegments

在 kafka.tools 包中还有一些其他工具类, 这些工具类在 \$KAFKA_HOME/bin 目录下并没有对应的脚本, 例如, DumpLogSegments、JmxTool 等。管理人员可以通过前面描述的 kafka-run-class 脚本使用这些工具类。

DumpLogSegments 工具类的主要功能是将指定的日志文件和索引文件中的内容打印到控制台, 它还可以实现验证索引文件的功能。DumpLogSegments.main() 方法的实现如下:

```

def main(args: Array[String]) {
    ..... // 验证并解析参数 (略)

    // 当索引项在对应的日志文件中找不到对应的消息时,会将其记录到misMatchesForIndexFilesMap
    // 集合中
    // 其中 key 为索引文件的绝对路径, value 是索引项中的相对 offset 和消息的 offset 组成的
    // 元组的集合
    val misMatchesForIndexFilesMap = new mutable.HashMap[String, List[(Long,
Long)]]

    // 如果消息是未压缩的,则需要 offset 是连续的,若不连续,则记录到
    // nonConsecutivePairsForLogFilesMap 集合中,其中 key 为日志文件的绝对路径,
    // value 是出现不连续
    // 消息的前后两个 offset 组成的元组的集合
    val nonConsecutivePairsForLogFilesMap =
        new mutable.HashMap[String, List[(Long, Long)]]

    for (arg <- files) {
        // 处理命令参数指定的文件集合
        val file = new File(arg)
        if (file.getName.endsWith(Log.LogFileSuffix)) {
            // 打印日志文件
            dumpLog(file, print, nonConsecutivePairsForLogFilesMap,
                isDeepIteration, maxMessageSize, messageParser)
        } else if (file.getName.endsWith(Log.IndexFileSuffix)) {
            // 打印索引文件
            dumpIndex(file, indexSanityOnly, verifyOnly,
                misMatchesForIndexFilesMap, maxMessageSize)
        }
    }
    ..... // 遍历 misMatchesForIndexFilesMap, 输出错误信息 (略)
    ..... // 遍历 nonConsecutivePairsForLogFilesMap, 输出错误信息 (略)
}

```

DumpLogSegments.dumpLog() 方法会遍历日志文件并打印消息的相关信息,例如,消息的 offset、position、压缩方式、CRC 校验码等,也会解析消息的 key 和 value 并打印。


```

private def dumpLog(file: File, printContents: Boolean,
    nonConsecutivePairsForLogFilesMap: mutable.HashMap[String,
    List[(Long, Long)]],
    isDeepIteration: Boolean, maxMessageSize: Int, parser:
    MessageParser[_]) {
    ... .. // 打印日志文件的 baseOffset (略)
    val messageSet = new FileMessageSet(file, false) // 创建 FileMessageSet 对象
    var validBytes = 0L // 记录通过验证的字节数
    var lastOffset = -1L // 记录 offset
    val shallowIterator = messageSet.iterator(maxMessageSize)
    for (shallowMessageAndOffset <- shallowIterator) { // 遍历日志文件中的消息
        // 根据 deep-iteration 参数以及消息是否压缩得到合适的迭代器
        val itr = getIterator(shallowMessageAndOffset, isDeepIteration)
        for (messageAndOffset <- itr) {
            val msg = messageAndOffset.message
            if (lastOffset == -1) // 记录上次循环处理的消息的 offset
                lastOffset = messageAndOffset.offset
            else if (msg.compressionCodec == NoCompressionCodec &&
                messageAndOffset.offset != lastOffset + 1) {
                // 如果消息是未压缩的, 则需要 offset 是连续的, 若不连续, 则需要记录
                var nonConsecutivePairsSeq = nonConsecutivePairsForLogFilesMap
                    .getOrElse(file.getAbsolutePath, List[(Long, Long)]())
                nonConsecutivePairsSeq ::= (lastOffset, messageAndOffset.offset)
                nonConsecutivePairsForLogFilesMap.put(file.getAbsolutePath,
                    nonConsecutivePairsSeq)
            }
            lastOffset = messageAndOffset.offset

            // 输出消息的相关信息
            print("offset: " + messageAndOffset.offset + " position: " +
            validBytes
            + " isvalid: " + msg.isValid + " payloadsize: " + msg.payloadSize
            + " magic: " + msg.magic + " compresscodec: " + msg.
            compressionCodec
            + " crc: " + msg.checksum)
            if (msg.hasKey)
                print(" keysize: " + msg.keySize)
            if (printContents) {

```



```

        val (key, payload) = parser.parse(msg) // 解析消息
        key.map(key => print(s" key: ${key}")) // 输出消息的 key
        // 输出消息的 value
        payload.map(payload => print(s" payload: ${payload}"))
    }
    println()
}
// 记录通过验证, 正常打印的字节数
validBytes += MessageSet.entrySize(shallowMessageAndOffset.message)
}
val trailingBytes = messageSet.sizeInBytes - validBytes
if (trailingBytes > 0) // 当发现验证失败的消息时, 输出提示信息
    println("...")
}

```

`DumpLogSegments.dumpIndex()` 方法会遍历索引文件, 并检测 Index 中的索引项是否能在对应的日志文件中找到对应消息, 它会根据 `verifyOnly` 参数决定是否打印索引项的内容。

```

private def dumpIndex(file: File, indexSanityOnly: Boolean, verifyOnly: Boolean,
    misMatchesForIndexFilesMap: mutable.HashMap[String, List[(Long, Long)]],
    maxMessageSize: Int) {
    val startOffset = file.getName().split("\\.")(0).toLong // 获取 baseOffset
    // 获取对应的日志文件
    val logFile = new File(file.getAbsolutePath.getParent,
        file.getName.split("\\.")(0) + Log.LogFileSuffix)
    val messageSet = new FileMessageSet(logFile, false) // 创建 FileMessageSet
    // 创建 OffsetIndex
    val index = new OffsetIndex(file, baseOffset = startOffset)
    // 对索引文件做简单检查, 例如: 最后一个索引项的 offset 大于 baseOffset,
    // 索引文件大小是 8 的倍数
    if (indexSanityOnly) {
        index.sanityCheck
        return
    }
}

```

```

for (i <- 0 until index.entries) {
    val entry = index.entry(i) // 读取索引项
    // 获取一个分片 FileMessageSet, 分片的起始位置是索引项指定的位置
    val partialFileMessageSet = messageSet.read(entry.position,
maxMessageSize)
    // 从分片 FileMessageSet 中获取第一条消息
    val messageAndOffset = getIterator(partialFileMessageSet.head,
isDeepIteration = true).next()
    if (messageAndOffset.offset != entry.offset + index.baseOffset) {
        // 如果消息的 offset 与索引项的 offset 不匹配, 则需要记录下来
        var misMatchesSeq = misMatchesForIndexFilesMap.getOrElse(file.
getAbsolutePath,
            List[(Long, Long)]())
        misMatchesSeq ::= (entry.offset + index.baseOffset, messageAndOffset.
offset)
        misMatchesForIndexFilesMap.put(file.getAbsolutePath, misMatchesSeq)
    }
    if (entry.offset == 0 && i > 0)
        return
    if (!verifyOnly) // 输出索引项的内容
        println("offset: %d position: %d".format(entry.offset + index.
baseOffset,
            entry.position))
}
}

```

5.9 kafka-producer-perf-test 脚本

kafka-producer-perf-test 脚本主要负责测试生产者的各种性能指标, 底层通过调用 ProducerPerformance 实现。从 Kafka 0.9.0 版本开始, 废弃了原有 core 模块中的 kafka.tools.ProducerPerformance, 开始使用 tools 模块中的 org.apache.kafka.tools.ProducerPerformance。ProducerPerformance.main() 方法是 kafka-producer-perf-test 脚本的入口函数。

```

public static void main(String[] args) throws Exception {
    // Argparse4 是 Python argparse 命令行解析器的 Java 版本, 在新版本的
    // ProducerPerformance 中
    // 使用此工具包完成命令行参数的解析和验证
    ArgumentParser parser = argparse();
    try {
        Namespace res = parser.parseArgs(args);
        .....// 解析命令行参数 (略)
        // 创建 KafkaProducer
        KafkaProducer<byte[], byte[]> producer =
            new KafkaProducer<byte[], byte[]>(props);
        // 根据 record-size 参数创建测试消息的负载
        byte[] payload = new byte[recordSize];
        Random random = new Random(0);
        for (int i = 0; i < payload.length; ++i)
            // 生成随机字节填充消息负载
            payload[i] = (byte) (random.nextInt(26) + 65);

        // 创建 ProducerRecord, topicName 通过 topic 参数指定
        ProducerRecord<byte[], byte[]> record =
            new ProducerRecord<>(topicName, payload);
        // 创建 Stats 对象, 用于各个指标的统计, 其中 num-records 参数指定了产生的消息个数
        Stats stats = new Stats(numRecords, 5000);
        long startMs = System.currentTimeMillis();
        ..... // 省略限流器的相关代码 (略)
        for (int i = 0; i < numRecords; i++) {
            long sendStartMs = System.currentTimeMillis();
            Callback cb = stats.nextCompletion(sendStartMs, payload.
length, stats);
            producer.send(record, cb); // 发送消息, 统计的相关工作在 Callback 中完成
            ..... // 省略限流器的相关代码 (略)
        }
        producer.close();
        stats.printTotal(); // 打印统计信息
    } catch (ArgumentParserException e) {
        ..... // 异常处理 (略)
    }
}

```


Stats 负责记录多项数据并在测试完成后输出测试的各项性能指标。Stats 中各个字段的含义如下所述。

- start: 记录开始测试时间戳。
- reportingInterval: 保存两次输出之间的时间间隔。
- sampling: 样本个数。样本的个数与指定发送的消息数量有关，默认是 500000 为一个样本。sampling 字段的初始化代码片段如下：

```
this.sampling = (int) (numRecords / Math.min(numRecords, 500000));
```

- latencies: 记录每个样本中的延迟。latencies 字段的初始化如下：

```
this.latencies = new int[(int) (numRecords / this.sampling) + 1];
```

- iteration: 记录迭代次数，与抽样有关。
- count: 记录整个过程中发送的消息个数。
- bytes: 记录发送消息的总字节数。
- maxLatency: 记录从消息发出到对应响应返回之间的延迟的最大值。
- totalLatency: 记录延迟的总时间。
- windowCount: 当前时间窗口中发送消息的个数。
- windowStart: 当前时间窗口的起始时间戳。
- windowMaxLatency: 记录当前时间窗口中最大的延时。
- windowTotalLatency: 记录当前时间窗口中延时的总时长。
- windowBytes: 记录当前窗口发送的总字节数。

Stats.record() 方法负责更新上述字段中的值，并按照 reportingInterval 字段指定的时间间隔打印统计信息。

```
public void record(int iter, int latency, int bytes, long time) {
    this.count++; // 计算发送消息个数
    this.bytes += bytes; // 计算发送总字节数
    this.totalLatency += latency; // 计算总延迟
    this.maxLatency = Math.max(this.maxLatency, latency); // 记录最大延迟
    this.windowCount++; // 计算当前窗口发送消息个数
```



```

        this.windowBytes += bytes; // 计算当前窗口发送总字节数
        this.windowTotalLatency += latency; // 计算当前窗口的总延迟
        // 记录当前窗口的最大延迟
        this.windowMaxLatency = Math.max(windowMaxLatency, latency);
        if (iter % this.sampling == 0) { // 选择样本, 更新 latencies 中对应值
            this.latencies[index] = latency;
            this.index++;
        }

        // 检测是否需要结束当前窗口, 并开启新窗口
        if (time - windowStart >= reportingInterval) {
            printWindow(); // 输出当前窗口中记录的信息
            newWindow(); // 清空 window* 字段, 开启下一个窗口的记录
        }
    }
}

```

在 `printWindow()` 方法中会将当前时间窗口内的消息个数、每秒发送的消息数、每秒发送的字节数（单位是 `mb`）、平均每个消息的延迟以及整个时间窗口的最大延迟等信息，按照指定格式输出到控制台。代码比较简单，就不贴出来了。

在 `main()` 方法中，发送的 `ProducerRecord` 对象都会创建一个对应的 `PerfCallback` 对象作为回调，在 `PerfCallback.onCompletion()` 中会调用 `Stats.record()` 方法记录相关的统计信息。

```

public void onCompletion(RecordMetadata metadata, Exception exception) {
    long now = System.currentTimeMillis();
    int latency = (int) (now - start); // 计算消息的延迟
    // 调用 Stats.record() 方法进行记录
    this.stats.record(iteration, latency, bytes, now);
    if (exception != null)
        exception.printStackTrace();
}

```

当全部的消息发送完之后，会调用 `Stats.printTotal()` 方法输出整个测试过程中全局的统计信息，例如，消息总个数、每秒发送的消息数、每秒发送的字节数（单位是 `mb`）、平均每个消息的延迟、最大延迟，还会计算延时时间的多个分位数。代码比较简单，就不贴出来了。

5.10 kafka-consumer-perf-test 脚本

kafka-consumer-perf-test 脚本负责测试消费者的各项性能指标，底层通过调用 ConsumerPerformance 实现，这里重点分析对新版本 Consumer 的性能测试。ConsumerPerformance.main() 方法是 kafka-consumer-perf-test 脚本的入口函数，具体实现如下：

```
def main(args: Array[String]): Unit = {
    val config = new ConsumerPerfConfig(args)
    val totalMessagesRead = new AtomicLong(0) // 从服务端拉取的消息数
    val totalBytesRead = new AtomicLong(0) // 获取消息的总字节数
    val consumerTimeout = new AtomicBoolean(false) // Consumer 是否超时
    ..... // 根据配置决定是否输出抬头信息（略）
    var startMs, endMs = 0L // 记录整个测试过程的开始时间戳和结束时间戳
    if (config.useNewConsumer) { // 新版本 Consumer
        // 创建 KafkaConsumer
        val consumer = new KafkaConsumer[Array[Byte], Array[Byte]](config.
props)
        consumer.subscribe(List(config.topic))
        startMs = System.currentTimeMillis
        consume(consumer, List(config.topic), config.numMessages, 1000, config,
            totalMessagesRead, totalBytesRead) // consume() 方法是测试的核心代码
        endMs = System.currentTimeMillis
        consumer.close()
    } else {
        .... // 旧版本 Consumer 的相关处理（略）
    }
    val elapsedSecs = (endMs - startMs) / 1000.0
    if (!config.showDetailedStats) {
        val totalMBRead = (totalBytesRead.get * 1.0) / (1024 * 1024)
        // 输出测试过程的开始时间、结束时间、消费的总字节数（单位是 MB）、每秒拉取的字节数、
        // 每秒消费的消息条数以及消费的总消息数
        println("%s, %s, %.4f, %.4f, %d, %.4f".format(
            config.dateFormat.format(startMs), config.dateFormat.
format(endMs),
            totalMBRead, totalMBRead / elapsedSecs, totalMessagesRead.get,
            totalMessagesRead.get / elapsedSecs))
    }
}
```

`ConsumerPerformance.consume()` 方法在开始测试之前会等待 Rebalance 操作完成，然后通过 `KafkaConsumer.poll()` 从服务端拉取消息并记录消息的大小、拉取时间等信息，之后间隔指定时间输出一组统计信息，最后更新 `totalMessagesRead` 和 `totalBytesRead` 用于 `main()` 方法最后的汇总输出。

```
def consume(consumer: KafkaConsumer[Array[Byte], Array[Byte]],
            topics: List[String], count: Long,
            timeout: Long, config: ConsumerPerfConfig,
            totalMessagesRead: AtomicLong, totalBytesRead: AtomicLong) {
    var bytesRead = 0L // 记录读取到的消息的总字节数
    var messagesRead = 0L // 记录拉取的消息个数
    var lastBytesRead = 0L
    var lastMessagesRead = 0L
    val joinTimeout = 10000 // 等待 Rebalance 操作完成的最长时间
    val isAssigned = new AtomicBoolean(false) // 标识当前消费者是否已经分配了分区

    // 订阅 Topic, 同时添加 ConsumerRebalanceListener 用来修改 isAssigned 的值
    consumer.subscribe(topics, new ConsumerRebalanceListener {
        def onPartitionsAssigned(partitions: util.Collection[TopicPartition]) {
            isAssigned.set(true)
        }

        def onPartitionsRevoked(partitions: util.Collection[TopicPartition]) {
            isAssigned.set(false)
        }
    })

    val joinStart = System.currentTimeMillis()
    // 循环等待分区分配过程完成
    while (!isAssigned.get()) {
        if (System.currentTimeMillis() - joinStart >= joinTimeout) {
            throw new Exception("Timed out waiting for initial group join.")
        }
        consumer.poll(100)
    }

    // 调整消费者, 从分区第一条消息开始消费
    consumer.seekToBeginning(List[TopicPartition]())
    // 下面开始正式的性能测试
```



```

val startMs = System.currentTimeMillis // 记录测试开始时间
val lastReportTime: Long = startMs
val lastConsumedTime = System.currentTimeMillis // 记录最后一次拉取消息时间

while (messagesRead < count &&
      System.currentTimeMillis() - lastConsumedTime <= timeout) {
    val records = consumer.poll(100)
    if (records.count() > 0)
        lastConsumedTime = System.currentTimeMillis
    for (record <- records) {
        messagesRead += 1 // 增加消费的总消息数量
        if (record.key != null)
            bytesRead += record.key.size // 增加消费的总字节数
        if (record.value != null)
            bytesRead += record.value.size // 增加消费的总字节数
    }
    // 间隔 reportingInterval 时间, 输出一次统计数据
    if (messagesRead % config.reportingInterval == 0) {
        if (config.showDetailedStats)
            printProgressMessage(0, bytesRead, lastBytesRead, messagesRead,
                                lastMessagesRead, lastReportTime, System.currentTimeMillis,
                                config.dateFormat)
        lastReportTime = System.currentTimeMillis // 记录输出时间
        lastMessagesRead = messagesRead // 更新, 供下次输出使用
        lastBytesRead = bytesRead
    }
}

totalMessagesRead.set(messagesRead) // 更新 totalMessagesRead
totalBytesRead.set(bytesRead) // 更新 totalBytesRead
}

```

5.11 kafka-mirror-maker 脚本

kafka-mirror-maker 脚本的主要功能是实现两个 Kafka 集群中的数据同步, 它的基本原理是通过消费者从源集群中获取消息, 再通过生产者将消息追加到目的集群中, 这样就实现了数据在两个集群中的迁移。kafka-mirror-maker 脚本通过调用 MirrorMaker 实现,

MirrorMaker 中支持新版本的消费者和生产者。

MirrorMaker 的结构图如图 5-5 所示, 其中创建了多个 MirrorMakerNewConsumer 对象, 每个 MirrorMakerNewConsumer 对象对应创建一个 MirrorMakerThread 线程, 实现从源集群中的消息拉取。多个 MirrorMakerThread 线程共享一个 MirrorMakerProducer 对象, 实现向目的集群追加消息的功能。

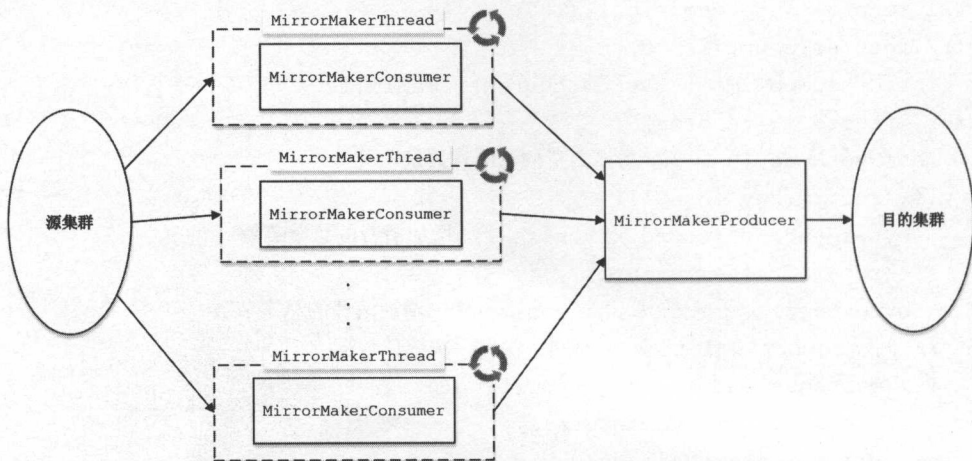


图 5-5

MirrorMaker 中各个字段的含义如下所述。

- producer: MirrorMakerProducer 对象。
- mirrorMakerThreads: MirrorMakerThread 线程集合。
- numDroppedMessages: AtomicInteger 对象, 记录 MirrorMaker 发送失败的消息个数。
- messageHandler: MirrorMakerMessageHandler 对象, 当 MirrorMakerNewConsumer 从源集群中将消息拉取到本地后, 会先经过 MirrorMakerMessageHandler 处理, 再发送给目的集群。此字段的具体类型是由 message.handler 参数指定的, 默认值是 defaultMirrorMakerMessageHandler。
- offsetCommitIntervalMs: 指定生产者进行 flush() 操作和提交 offset 的周期。

MirrorMaker.main() 是 kafka-mirror-maker 脚本的入口方法, 首先会使用 joptsimple 解析并验证命令行参数, 设置 JVM 关闭钩子, 然后创建 MirrorMakerProducer 和 MirrorMakerNewConsumer, 并为每个 MirrorMakerNewConsumer 创建对应的 MirrorMakerThread 线程, 之后创建并初始化 MessageHandler, 最后启动

MirrorMakerThread 线程，主线程阻塞等待全部的 MirrorMakerThread 线程结束。

```
def main(args: Array[String]) {
  .....// 解析命令行参数 (略)
  Runtime.getRuntime.addShutdownHook(new Thread("MirrorMakerShutdownHook"))
  {
    override def run() { // 添加 JVM 关闭钩子
      cleanShutdown()
    }
  })
  .....// 获取生产者相关的配置 (略)
  producer = new MirrorMakerProducer(producerProps) // 创建MirrorMakerProducer
  val mirrorMakerConsumers = // 创建消费者
  if (!useNewConsumer) {
    ..... // 创建旧版本消费者 (略)
  } else {
    .....// 根据 consumer.rebalance.listener 参数决定是否创建 ConsumerRebalanceListener (略)
    .....// 检测 customRebalanceListener 的类型是否正确, 若不正确则抛出异常 (略)
    createNewConsumers(
      numStreams, // 消费者个数
      options.valueOf(consumerConfigOpt), // 消费者的相关配置
      customRebalanceListener,
      Option(options.valueOf(whitelistOpt)) // 白名单
    )
  }

  // 创建 MirrorMakerThread 线程, 与消费者一一对应
  mirrorMakerThreads = (0 until numStreams) map (i =>
    new MirrorMakerThread(mirrorMakerConsumers(i), i))

  // 根据 message.handler 参数创建并初始化 MessageHandler
  val customMessageHandlerClass = options.valueOf(messageHandlerOpt)
  val messageHandlerArgs = options.valueOf(messageHandlerArgsOpt)
  messageHandler = {
    if (customMessageHandlerClass != null) {
      if (messageHandlerArgs != null)
        CoreUtils.createObject[MirrorMakerMessageHandler]
          (customMessageHandlerClass, messageHandlerArgs)
    }
  }
}
```

```

        else
            CoreUtils.createObject[MirrorMakerMessageHandler](customMessageHandlerClass)
        } else {
            defaultMirrorMakerMessageHandler
        }
    }
    mirrorMakerThreads.foreach(_.start()) // 启动 MirrorMakerThread 线程
    // 主线程阻塞, 等待 MirrorMakerThread 线程全部结束
    mirrorMakerThreads.foreach(_.awaitShutdown())
}

```

MirrorMakerProducer 是对 KafkaProducer 的简单封装。这里要注意, 在 MirrorMakerProducerCallback 中会统计发送失败的消息数量, 还会根据 abort.on.send.failure 配置决定在出现发送失败的情况时是否关闭生产者。具体实现如下:

```

private class MirrorMakerProducer(val producerProps: Properties) {
    val sync = producerProps.getProperty("producer.type", "async").
equals("sync")
    val producer = new KafkaProducer(Array[Byte], Array[Byte])(producerProps)

    def send(record: ProducerRecord(Array[Byte], Array[Byte])) {
        if (sync) { // 同步发送
            this.producer.send(record).get()
        } else { // 异步发送
            this.producer.send(record, new MirrorMakerProducerCallback(record.
topic(),
                record.key(), record.value()))
        }
    }
    ..... // flush() 方法、close() 方法都是通过直接调用 KafkaProducer 对应的方法实现(略)
}

// 下面是 MirrorMakerProducerCallback.onCompletion() 方法的实现
override def onCompletion(metadata: RecordMetadata, exception: Exception) {
    if (exception != null) { // 下面是出现异常的相关处理
        super.onCompletion(metadata, exception) // 通过父类的实现输出错误日志
    }
}

```



```
// 如果设置了 abort.on.send.failure 参数, 则停止 MirrorMaker, 否则忽略异常, 继续发送后面的消息
if (abortOnSendFailure) {
    info("Closing producer due to send failure.")
    // 设置为 true 后会通知全部 MirrorMakerThread 停止
    exitingOnSendFailure = true
    producer.close(0)
}
numDroppedMessages.incrementAndGet() // 记录发送失败的消息数量
}
```

MirrorMaker.createNewConsumers() 方法负责创建多个 MirrorMakerNewConsumer, 具体数目由 num.streams 参数指定。

```
def createNewConsumers(numStreams: Int, consumerConfigPath: String,
    customRebalanceListener: Option[ConsumerRebalanceListener],
    whitelist: Option[String]) : Seq[MirrorMakerBaseConsumer] = {
    ..... // 获取 KafkaConsumer 的相关配置项 (略)
    // 创建指定数量的 KafkaConsumer 对象
    val consumers = (0 until numStreams) map { i =>
        consumerConfigProps.setProperty("client.id", groupIdString + "-" +
i.toString)
        new KafkaConsumer(Array[Byte], Array[Byte])(consumerConfigProps)
    }
    // 检测是否指定白名单, 白名单指明了需要进行镜像的 Topic
    whitelist.getOrElse(throw new IllegalArgumentException("..."))
    // 创建 MirrorMakerNewConsumer
    consumers.map(consumer => new MirrorMakerNewConsumer(consumer,
        customRebalanceListener, whitelist))
}
```

MirrorMakerNewConsumer 通过 KafkaConsumer 实现从源集群中拉取消息的功能。在 MirrorMakerNewConsumer 实现中关闭了 KafkaConsumer 自动提交 offset 的功能, 而使用 offsets 字段 (HashMap 类型) 维护其消费的分区的 offset, 并在进行 Rebalance 操作前通过 InternalRebalanceListenerForNewConsumer 提交 offset。


```

private class MirrorMakerNewConsumer(consumer: Consumer[Array[Byte],
Array[Byte]],
    customRebalanceListener: Option[ConsumerRebalanceListener],
    whitelistOpt: Option[String]) extends MirrorMakerBaseConsumer {
    var recordIter: Iterator[ConsumerRecord[Array[Byte], Array[Byte]]] =
null
    // 在MirrorMakerNewConsumer中使用offsets字段自己维护其消费的offset
    private val offsets = new HashMap[TopicPartition, Long]()

    override def init() {
        val consumerRebalanceListener = new InternalRebalanceListenerForNewCon
sumer(
            this, customRebalanceListener) // 创建InternalRebalanceListenerForNewConsumer
        if (whitelistOpt.isDefined) {
            try {
                // 订阅白名单指定的Topic
                consumer.subscribe(Pattern.compile(whitelistOpt.get),
                    consumerRebalanceListener)
            } catch {
                ... .. // 异常处理(略)
            }
        }
    }

    override def receive(): BaseConsumerRecord = {
        if (recordIter == null || !recordIter.hasNext) {
            recordIter = consumer.poll(1000).iterator // 从源集群中拉取消息
            if (!recordIter.hasNext)
                throw new ConsumerTimeoutException
        }
        val record = recordIter.next()
        val tp = new TopicPartition(record.topic, record.partition)
        offsets.put(tp, record.offset + 1) // 记录消费的offset
        // 返回从源集群中拉取的消息
        BaseConsumerRecord(record.topic, record.partition, record.offset,
            record.timestamp, record.timestampType, record.key, record.value)
    }
}

```

```
// stop() 方法、cleanup() 方法、commit() 方法等都是通过直接调用 KafkaConsumer 的
// 对应方法完成的（略）
}
```

InternalRebalanceListenerForNewConsumer 实现了 ConsumerRebalanceListener 接口，在进行 Rebalance 操作之前会将 offsets 字段中记录的 offset 进行提交。

```
private class InternalRebalanceListenerForNewConsumer(mirrorMakerConsumer:
    MirrorMakerBaseConsumer, customRebalanceListenerForNewConsumer:
    ConsumerRebalanceListener) extends ConsumerRebalanceListener {

    override def onPartitionsRevoked(partitions: Collection[TopicPartition])
    {
        producer.flush() // 将 KafkaProducer 缓存的消息全部发送到目的集群
        commitOffsets(mirrorMakerConsumer) // 提交 offsets 集合中记录的 offset
        // 调用 consumer.rebalance.listener 参数指定的 ConsumerRebalanceListener 实
        // 现自定义的功能
        customRebalanceListenerForNewConsumer.foreach(_.onPartitionsRevoked(pa
            rtitions))
    }

    override def onPartitionsAssigned(partitions: util.Collection[TopicPartition]) {
        customRebalanceListenerForNewConsumer.foreach(_.onPartitionsAssigned
            (partitions))
    }
}
```

MirrorMakerThread 线程会首先调用对应 MirrorMakerNewConsumer 对象的 receive() 方法获取消息，然后将消息交给 MirrorMakerMessageHandler.handle() 方法处理，之后将处理完的消息追加到目的集群中。在这个过程中会按照 offset.commit.interval.ms 参数指定的时间间隔提交 offset。

```

override def run() {
  try {
    mirrorMakerConsumer.init() // 初始化MirrorMakerNewConsumer
    while (!exitingOnSendFailure && !shuttingDown) {
      try {
        // 检测 exitingOnSendFailure、shuttingDown 等标记
        while (!exitingOnSendFailure && !shuttingDown && mirrorMakerConsumer.
hasData) {
          val data = mirrorMakerConsumer.receive() // 从源集群中获取消息
          // 通过 MirrorMakerMessageHandler 创建 ProducerRecord
          val records = messageHandler.handle(data)
          records.foreach(producer.send) // 发送消息
          maybeFlushAndCommitOffsets() // 尝试提交 offset
        }
      } catch {
        ..... // 出现异常时输出日志 (略)
      }
      maybeFlushAndCommitOffsets()
    }
  } catch {
    .....// 出现异常时输出日志 (略)
  } finally {
    .....// 关闭 Producer 和 Consumer、提交 offset、尝试唤醒主线程 (略)。
  }
}

// 下面是 MirrorMakerThread.maybeFlushAndCommitOffsets() 方法的实现
def maybeFlushAndCommitOffsets() {
  // 满足 offset.commit.interval.ms 参数指定的时间间隔, 才提交一次 offset
  if (System.currentTimeMillis() - lastOffsetCommitMs > offsetCommitIntervalMs) {
    producer.flush() // 将 KafkaProducer 缓冲的消息发送出去
    // 提交 MirrorMakerNewConsumer.offsets 集合中记录的 offset
    commitOffsets(mirrorMakerConsumer)
    lastOffsetCommitMs = System.currentTimeMillis() // 记录最近一次提交时间戳
  }
}
}

```

kafka-mirror-maker 脚本也支持旧版本的消费者, 这里不做详细分析, 感兴趣的读者

可以参考源码进行学习。

本章小结

在 Kafka 中提供了一些实现常见功能的脚本，这些脚本可以帮助管理人员快速完成一些常见的管理、运维、测试功能。通过本章的分析，配合 Kafka 官方网站的相关文档和参数说明，希望读者能够了解这些脚本工具的具体实现和运行原理，在实际工作中更好地使用这些脚本。

参考文献

参考书籍

- [1] BruceEckel. Java 编程思想: 第 4 版 [M]. 机械工业出版社, 2007.
- [2] DeanWampler, AlexPayne, 万普勒, 等. Scala 编程 [M]. 东南大学出版社, 2015.
- [3] Martin Odersky, Lex Spoon, Bill Venners. Scala 编程 [M]. 电子工业出版社, 2010.
- [4] ErichGamma, 加马, 李英军. 设计模式: 可复用面向对象软件的基础 [J]. 2000.

参考网络资源

Kafka 官方文档: <http://kafka.apache.org/documentation/>

confluent 官方博客: <https://www.confluent.io/blog/>

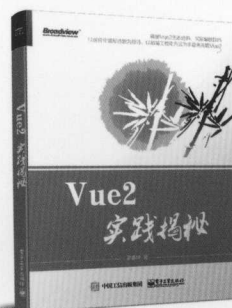
Scalable IO in Java: <http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf>

Apache 基金会 Kafka 相关 Wiki: <https://cwiki.apache.org/confluence/collector/pages.action?key=KAFKA>

fxjwind 的博客: <http://www.cnblogs.com/fxjwind/>

lizhitao 的博客: <http://blog.csdn.net/lizhitao>

chunlongyu 的博客: <http://blog.csdn.net/chunlongyu>



Apache Kafka 源码剖析

专家力荐

《Apache Kafka源码剖析》一书深入浅出地分析了Kafka的源代码，无论是刚接触Kafka的菜鸟，还是已经有多年Kafka使用经验的老鸟，这本书都能让你有所收获。

——搜狗高级研发工程师 张亚森

Kafka是大数据平台中的关键部分之一。《Apache Kafka源码剖析》全面细致地剖析了Kafka的运行原理和架构设计，在带领读者进入Kafka源码世界的同时，也分析了许多设计经验，是一本不可多得的好书。

——华为高级研发工程师 张占龙

在阅读《Apache Kafka源码剖析》时，作者在每一章节中都会给我意外之惊喜。作者对Kafka源代码已有相当深刻的理解，此书代码分析过程逻辑清晰，详略得当，实属不易。

——网易游戏高级数据挖掘研究员 杨威

大型分布式系统犹如一个生命，系统中各个服务犹如骨骼，其中的数据犹如血液，而Kafka犹如经络，串联整个系统。《Apache Kafka源码剖析》通过大量的设计图展示、代码分析、示例分享，把Kafka的实现脉络展示在读者面前，帮助读者更好地研读Kafka代码。

——今日头条高级研发工程师 刘克刚

《Apache Kafka源码剖析》中汇集了作者多年Kafka开发经验，为读者深入学习Kafka实现指明了方向。对于想学习Kafka的程序员来说，这是一本非常不错的进阶书籍。

——美团高级研发工程师 刘思



博文视点Broadview



@博文视点Broadview



责任编辑：陈晓猛
封面设计：王 乐

上架建议：计算机>分布式系统

ISBN 978-7-121-31345-5



9 787121 313455 >

定价：89.00元